Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Domen Kos

Primerjava načrtovanja in izvedbe poslovnega procesa z uporabo modelov BPM in UML

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: viš. pred. dr. Igor Rožanc

Ljubljana, 2019

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Pred razvojem zahtevnejših poslovnih aplikacij je smiselno le-te ustrezno modelirati. Model poslovnega procesa omogoča preglednejši razvoj aplikacije, ki jo je deloma mogoče tudi avtomatsko generirati. V diplomskem delu predstavite razvoj poslovne aplikacije z uporabo tovrstnega prostopa. V ta namen najprej predstavite BPM in UML notaciji za opis modela poslovnega procesa ter izberite primernejšo. To uporabite za opis poslovnega procesa likvidacije vhodnega računa ter iz modela avtomatsko generirajte ogrodje delujoče rešitve. Manjajoči del rešitve dopolnite z uporabo knjižnice Boost MSM. Nalogo zaključite z prikazom in analizo dobljene rešitve.

Iskreno se zahvaljujem vsem, ki so me podpirali in mi pomagali tekom študija.

Kazalo

Slike

Seznam uporabljenih kratic

kratica	angleško	slovensko
UML	Unified Modeling Language	Poenoten jezik modeliranja
UML AD	UML Activity Diagram	UML diagram aktivnosti
BPMN	Buisness Process Modeling Notation	Notacija za modeliranje poslovnih procesov
BPEL	Buisness Process Execution Langugae	Jezik za izvajanje poslovnih procesov
OOP	Object-Oriented Programming	Objektno usmerjeno programiranje
MDD	Model Driven Development	Modelno usmerjen razvoj
Boost MSM	Boost Meta State Machine	Modul iz knjižnice Boost
GUI	Graphical User Interface	Grafični uporabniški vmesnik
MVC	Model-View-Controller	Model-pogled-nadzornik
API	Application Programming Interface	Aplikativni programski vmesnik

Povzetek

Naslov: Primerjava načrtovanja in izvedbe poslovnega procesa z uporabo modelov BPM in UML

Avtor: Domen Kos

V diplomski nalogi je predstavljena problematika načrtovanja in izvedbe poslovnih procesov z uporabo specifičnega pristopa. Pri tem smo pokrili celotno pot od definiranja poslovnega procesa do končne izvedbe in poganjanja le-tega. Poslovni proces smo definirali z uporabo ustrezne grafične notacije, saj vizualna predstavitev prinaša pomembne prednosti. Z izbrano notacijo smo predstavili model in ga delno avtomatizirano pretvorili v programsko kodo. Kjer to ni bilo mogoče, smo si pomagali z modulom MSM iz knjižnice Boost.

Predstavili smo splošno namenski modelirni jezik UML, kjer smo večjo pozornost posvetili diagramu aktivnosti in diagramu stanj. Nato smo raziskali še notacijo BPM, ki je namenjena načrtovanju poslovnih procesov. Ob predstavitvi obeh smo ugotovili, da ima UML diagram aktivnosti največ skupnih točk z notacijo BPM. Med seboj smo ju primerjali in se na podlagi ugotovitev odločili za uporabo UML diagrama aktivnosti.

Z namenom prikaza načrtovanja poslovnega procesa smo definirali poslovni proces likvidacije vhodnega računa in ga modelirali z uporabo diagrama aktivnosti. Z uporabo razrednega diagrama smo definirali objekte, ki nastopajo v procesu in za njih generirali programsko kodo C++.

S pomočjo modula MSM, ki temelji na diagramih stanj, smo implementirali proces likvidacije vhodnega računa. Ob izvedbi je podrobneje predsta-

vljen tudi sam modul MSM.

Glavni rezultat diplomske naloge je delujoča aplikacija, ki simulira vhodne akcije različnih uporabnikov in vodi proces likvidacije vhodnega računa. V zaključku naloge je predstavljena še analiza dobljenih rezultatov.

Ključne besede: modeliranje poslovnih procesov, UML, BPMN, Boost MSM.

Abstract

Title: Comparison of design and implementation of business process using BPM and UML models

Author: Domen Kos

The thesis presents the problem of designing and implementation of business processes using specific approach. We presented the whole path from defining the process to the final implementation and execution. The business process is defined by a graphic notation, since the visual presentation of data brings important advantages. The selected notation is used to model the process and convert it to source code as automatically as possible. Where automatic conversion was not possible, we used module MSM from Boost Library.

We presented a general purpose model language UML, with the emphasis on activity diagram and state diagram. The BPMN standard for modeling business processes was introduced as well. BPMN has several common points with UML activity diagram. After the comparison we chose to use UML activity diagram for further purposes.

To demonstrate the design and implementation of business processes, we defined the business process of the input invoice liquidation and modeled it using activity diagram. In addition, we used class diagram to define objects and generated the C++ code for them.

With the help of the MSM module which is based on state machines, we implemented the input invoice liquidation. In this part the module is presented as well.

The main thesis result is a test application which simulates the inputs of different users and leads the whole process of the input invoice liquidation. Finally the analysis of obtained result is presented in the final part of the thesis.

Keywords: business process modeling, UML, BPMN, Boost MSM.

Poglavje 1

Uvod

V dobi digitalizacije, v kateri razvijalci programske opreme vsakodnevno razvijajo aplikacije za uporabo tako v zasebnem kot poslovnem življenju, je čedalje večja potreba po tehnologijah, ki nam lajšajo vsakodnevna opravila. Večja pa je tudi vloga in uporabnost, ki jo želimo prenesti na takšne sisteme. Tako ti postajajo večji, zahtevnejši in prevzemajo večjo odgovornost v podjetju. Programska rešitev, ki to omogoča, je veliko več kot le skupek programske kode. Biti mora robustna, lahko prilagodljiva in ustrezati varnostnim standardom. Sestavljena mora biti tako, da jo lahko tudi razvijalci, ki so na projekt prišli kasneje, brez večjih težav dopolnijo ali odpravijo napako. Programsko kodo našega sistema lahko ponovno uporabimo na različnih delih sistema in nam je ni potrebno podvajati.

Ker si ljudje lažje predstavljamo stvari vizualno, se pri zahtevnejših sistemih uporablja različne postopke, katerih rezultat je prav vizualna predstavitev zasnovanega sistema. To so tako imenovani modeli (angl. models), ki našo infrastrukturo orisujejo iz različnih vidikov. Na ta način lahko že pri modeliranju nekega procesa zasnujemo sestavne dele ali pa potek izvajanja našega končnega programa.

Model sam po sebi ne zna ničesar. Potrebno ga je prevesti v programsko kodo in ga v tej obliki tudi poganjati. To lahko dosežemo na različne načine. Najpogosteje tega ni mogoče doseči popolnoma avtomatizirano in neodvisno

od orodja, s katerim smo izdelali model. Čeprav nekatera orodja zmorejo generirati dele programske kode za kasnejšo vključitev v končno rešitev, pa je do popolne izvedbe izrisanega modela marsikaj potrebno dodelati ročno.

V nadaljevanju bomo preučili pot od zasnove modela do pretvorbe končne različice tega v programsko kodo. Najprej bomo predstavili dva modelirna standarda, ki sta v praksi pogosto uporabljena, ju med seboj primerjali in izbrali primernejšega. Nato bomo natančno definirali poslovni proces za konkreten postopek likvidacije vhodnega računa ter ga zmodelirali z uporabo izbranega standarda. Končno različico modela bomo čim bolj avtomatizirano pretvorili v programsko kodo. Za dele, ki jih ne bo mogoče avtomatizirano pretvoriti, si bomo pomagali s programsko knjižnico Boost in modulom MSM. Omenjen modul bomo ob uporabi čim bolje preučili ter v analizi podali mnenje o možnostih uporabe le-tega v produkcijske namene.

Opisan postopek je pristop razvoja, ki se imenuje modelno usmerjen razvoj (angl. model driven developement). Gre za način razvoja, kjer je abstrakten model možno bolj ali manj avtomatizirano pretvoriti v programsko kodo, le-to pa nazaj v model. V analizi bomo to tehniko bolje predstavili in raziskali, ali je modul MSM primeren za tako vrsto razvoja.

Rezultat diplomskega dela bo ob predstavitvi primerjave obeh standardov in modula MSM tudi izvedljiva programska rešitev, ki bo simulirala delovni tok v definiranem poslovnem procesu.

Poglavje 2

Predstavitev modelirnega jezika UML in notacije BPM

V poglavju bomo predstavili modelirni jezik UML [22] in notacijo BPM [40]. Namen notacije BPM je modeliranje poslovnih procesov. UML je splošna notacija, vendar pa jo lahko uporabimo tudi za modeliranje poslovnih procesov. Iz tega vidika bomo več pozornosti posvetili predvsem UML diagramu aktivnosti, ki je po strukturi in simbolih najbolj podoben BPM notaciji. Bolj podrobno se bomo seznanili tudi z diagramom stanj, ki je prav tako eden izmed diagramov jezika UML. Modul MSM [2], ki bo uporabljen pri izvedbi, namreč za svoje delovanje uporablja prav diagrame stanj. Ob koncu poglavja bo na kratko predstavljen tudi modul MSM, njegovo uporabo pa bomo podrobneje predstavili ob programiranju zastavljene naloge.

2.1 Modelirni jezik UML

Modeliranje je tehnika, ki jo uporabimo preden začnemo pisati kodo [17]. Razvijalci jo običajno uporabljajo za predstavitev infrastrukture v vizualni obliki in ima za njih podobno vlogo, kot jo ima načrt (angl. blueprint) za arhitekta [16]. Če razvijalci modelirajo projekt preden ga začnejo kodirati, si lahko zagotovijo, da so v načrt vključili vse specifikacije in končne

uporabnike. Tak načrt ustreza tudi varnostnim standardom, je prilagodljiv ter robusten, po implementaciji pa služi tudi kot dokumentacija. Kasnejše spremembe ali popravki, ki so potrebni zaradi slabe zasnove, lahko stanejo podjetje veliko virov in mu prinesejo ogromne izgube.

Modelirni jezik UML (angl. Unified Modeling Language) [22, 24] je standard, ki vsebuje različne diagrame, ki razvijalcem pomagajo, da projekt vizualizirajo z različnih vidikov in se tako izognejo naštetim težavam. Prvotno je bil razvit za podporo načrtovalcev sistemov, programskim inženirjem in razvijalcem programske opreme, katerim je služil kot orodje za analizo, načrtovanje in izvedbo programsko zasnovanih sistemov. Uporablja pa se ga tudi v namen načrtovanja poslovnih in podobnih procesov [10, 16].

2.1.1 Zgodovina modelirnega jezika UML

V poznih osemdesetih prejšnjega stoletja in začetku devetdesetih se je skupaj z objektno usmerjenim pristopom programiranja pojavila tudi potreba po jeziku, ki bi razvijalcem in načrtovalcem programske opreme omogočal enostavnejše načrtovanje teh sistemov ter jih predstavila v obliki objektov, ki bi jih kasneje lahko pretvorili v objektno kodo v izbranem programskem jeziku. Ko sta leta 1994 začela s sodelovanjem Grady Booch in Jim Rumbaugh, ki sta se predhodno ukvarjala z iskanjem rešitve na omenjenem področju, je začel nastajati jezik UML. Leta 1995 se jima je pridružil še Ivar Jacobson, ki je prav tako raziskoval to področje. Njihova skupna rešitev se je imenovala OOSE metoda (angl. Object-Oriented Software Engineering method) [34].

Skupina raziskovalcev je moči združila iz različnih razlogov, vsak izmed njih pa je v skupino prinesel svoje znanje in izkušnje. Z dopolnjevanjem originalne metode z rezultati raziskav vsakega izmed njih je metoda postajala splošnejša in je tako razvijalcem prihranila čas pri učenju. Leta 1996 so izdali UML verzije 0.9 in kmalu so ga začela uporabljati tudi večja podjetja, ki so zagotovila tudi vire, da se je jezik stabiliziral in nadaljnjo razvijal. Tako je januarja leta 1997 skupina predstavila UML organizaciji OMG (angl. Object Management Group) z namenom, da je za jezik sprejet standard.

UML je obsežen jezik, ki ga v grobem delimo na diagrame obnašanja (angl. Behaviour diagrams) in strukturne diagrame (angl. structual diagrams) [42]. Vsako vrsto sestavlja večje število različnih diagramov, ki so našteti v nadaljevanju. Prikazani so tudi na sliki 2.1.

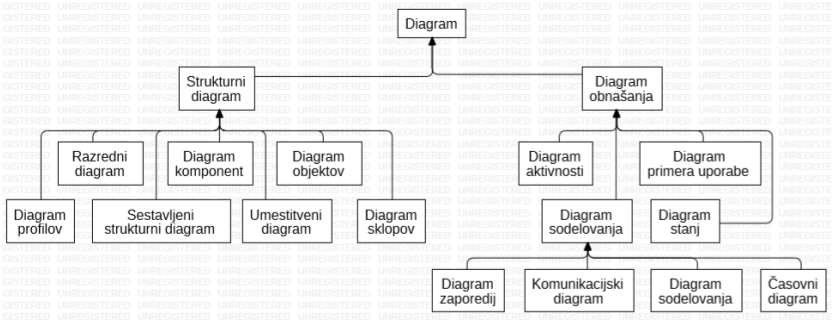
Diagrami obnašanja:

1. Diagram aktivnosti (angl. Activity Diagram)
2. Diagram primerov uporabe (angl. Use Case Diagram)
3. Diagram sodelovanja (angl. Interaction Overview Diagram)
4. Časovni diagram (angl. Timing Diagram)
5. Diagram stanj (angl. State Machine Diagram)
6. Diagram zaporedij (angl. Sequence Diagram)
7. Komunikacijski diagram (angl. Communication Diagram)

Strukturni diagrami:

1. Razredni diagram (angl. Class Diagram)
2. Diagram objektov (angl. Object Diagram)
3. Diagram komponent (angl. Component Diagram)
4. Sestavljen strukturni diagram (angl. Composite Structure Diagram)
5. Diagram sklopov (angl. Package Diagram)
6. Diagram profilov (angl. Profile Diagram)
7. Umestitveni diagram (angl. Deployment Diagram)

Vendar za uporabo UML standarda ni potrebno osvojiti vseh podrobnosti za vse diagrame. Poiskati je treba le vrsto diagrama, ki najbolj ustreza našim potrebam.



Slika 2.1: UML diagrami [21].

Na naš sistem lahko gledamo iz različnih vidikov, ki jih UML podpira. Če na projektu delajo različne skupine ljudi, kot so snovalci (angl. designers), koderji, testerji, podpora in drugi, lahko zanje najdemo ustrezen diagram. Za različne subjekte so primerni različni UML diagrami, ki prikazujejo različne perspektive istega sistema.

2.1.2 Strukturni diagrami

Strukturni diagrami prikazujejo statično strukturo sistema v različnih fazah razvoja. Gre za različne abstrakcije sistema, ki nam s povezovanjem posameznih delov oz. struktur združujejo sistem v celoto [22, 42, 24].

Diagram aktivnosti

Pri razvoju programske opreme se ta diagram uporablja za opis različnih aktivnosti in akcij skozi celoten proces, ki se lahko izvajajo zaporedno ali vzporedno ter si med seboj podajajo objekte vključene v proces (slika 2.2).

Aktivnost (angl. activity) predstavlja množico akcij.

Akcija (angl. action) opravilo, ki se izvede.

Tok izvajanja (angl. control flow) predpisuje vrstni red izvajanja aktivnosti.

Objektna povezava (angl. object flow) med seboj povezuje objekte, ki potujejo med aktivnostmi.

Objekt (angl. object node) predstavlja objekt v diagramu in je povezan z objektno povezavo.

Začetno stanje (angl. initial node) je vstopna točka v diagram.

Končno stanje (angl. final node) je izstopna točka iz diagrama.

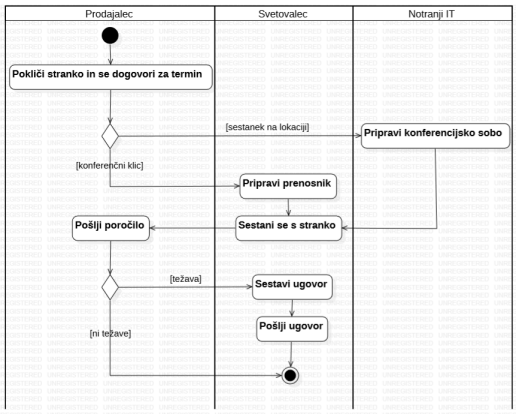
Odločitvena točka (angl. decision node) je razpotje v diagramu, kjer je vedno izbrana le ena pot.

Točka stikanja (angl. merge node) je točka, v kateri se spojijo poti, ki so nastale v odločitvenem vozlišču.

Razčlenitveno stanje (angl. fork node) aktivnost razdeli na več pod aktivnosti in jih izvede vzporedno.

Združitveno stanje (angl. join node) združi vzporedno izvajane aktivnosti.

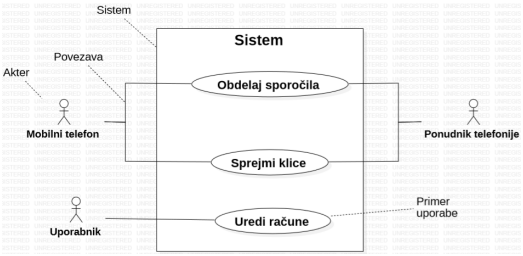
Plavalni pas (angl. swimlane) grupira aktivnosti po osebkih, ki jih izvajajo.



Slika 2.2: Diagram aktivnosti [25].

Diagram primerov uporabe

Ta diagram prikazuje funkcionalnosti sistema skozi primere. Je model, ki poleg opisa pričakovane akcije in opravila vključi še akterje (angl. actors), ki so pravzaprav uporabniki sistema – tako zunanji, (npr. kupec) kot notranji, (npr. prodajalec). Diagram nam torej predstavi, katere akcije so na voljo določenemu uporabniku v posameznem stanju, ter kakšne rezultate lahko pričakuje (slika 2.3).

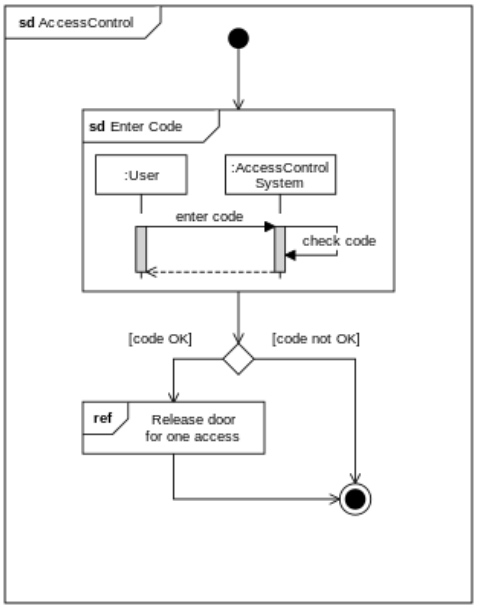


Slika 2.3: Diagram primera uporabe [32].

Diagram sodelovanja

To je ena izmed zahtevnejših vrst UML diagrama, ki nudi visok nivo abstrakcije. Vozlišča v diagramu so označena kot interakcije (angl. interactions) ali uporabniki teh interakcij (angl. interaction use). Diagram sodelovanja (slika 2.4) je zahtevnejša vrsta diagrama aktivnosti, ki je sestavljen iz več poddiagramov, ki so lahko:

- diagram sodelovanja,
- časovni diagram,
- diagram zaporedij ali
- komunikacijski diagram.



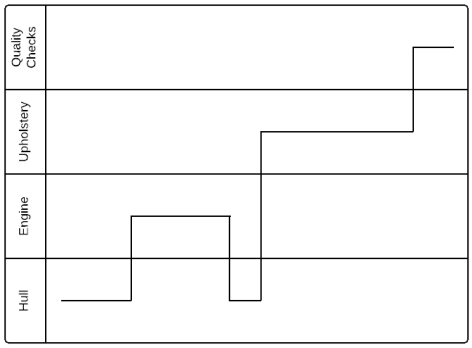
Slika 2.4: Diagram sodelovanja [30].

Časovni diagram

Ta diagram se uporablja, ko želimo predstaviti, kako se opazovani objekt obnaša skozi čas. V diagramu ne opazujemo, kako različni objekti komunicirajo med seboj, temveč, kako se ti spreminjajo s časom. Vsak izmed njih ima časovno premico, na kateri so akcije, ki ga spreminjajo. Primer časovnega diagrama je na sliki 2.5.

Diagram stanj

Diagram stanj uporabljamo za opis obnašanja opazovanega objekta v sistemu, ki je tesno odvisen od stanja, v katerem se nahaja [41]. Objekt se lahko na isto akcijo odzove drugače, v odvisnosti od stanja, v katerem se nahaja. Ti so lahko katerega koli tipa, morajo pa imeti definirane akcije, s katerimi komunicirajo z drugimi objekti v sistemu. To so lahko akterji, metode, podsistemi ali sistemi. Diagram stanj (slika 2.6) se lahko uporablja



Slika 2.5: Časovni diagram [23].

v kombinaciji z diagramom zaporedij, saj prvi opisuje vse akcije in dogodke enega samega objekta, drugi pa opisuje dogodke in akcije vseh objektov v sistemu za določeno iteracijo.

Diagram stanj vsebuje naslednje gradnike:

Začetno stanje (angl. initial state) je vstopna točka v diagram stanj.

Stanje (angl. state) je omejitev oziroma situacija v življenjskem ciklu opazovanega objekta, v katerem je omejitev izpolnjena. Opazovani objekt izvrši aktivnost ali čaka na dogodek.

Dogodek (angl. event) navadno povzroči spremembo v diagramu, kar lahko privede tudi v spremembo stanja. Prav tako imamo zunanje (angl. external) in notranje (angl. internal) dogodke, ki lahko vplivajo na celoten sistem ali pa samo na podsistem - diagram stanj. Dogodki lahko tudi prenašajo informacije med objekti. Ločimo štiri vrste dogodkov:

1. Signalni dogodek (angl. signal event): zgodi se ob prihodu signala ali sporočila.
2. Klicni dogodek (angl. call event): zgodi se ob klicu procedure.
3. Časovni dogodek (angl. time event): zgodi se, ko v podsistemu preteče določen čas.

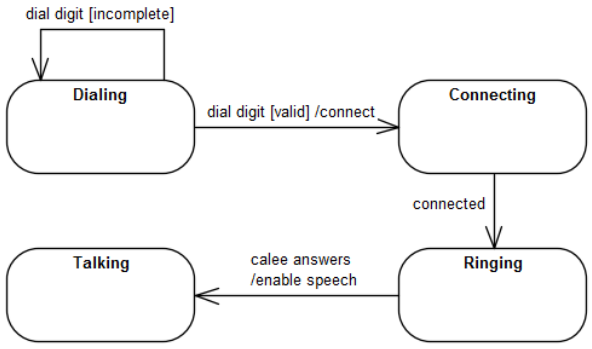
4. Dogodek spremembe (angl. change event): zgodi se vsakič, ko je izpolnjen pogoj.

Aktivnost (angl. activity) je vezana na stanje in je lahko končna ali pa se izvaja v neskončno. Aktivnost bo ustavljena takrat, ko se bo zgodil dogodek, ki bo sprožil prehod iz stanja, na katerega je vezana aktivnost, v neko drugo stanje.

Akcija (angl. action) je izvedljiva operacija, ki lahko vsebuje klice procedur, kreiranje ter uničenje drugega objekta ali pošiljanje signala objektom. Akcija je vezana na prehod in je ni mogoče prekiniti.

Prehod (angl. transition) se nahaja med dvema stanjema. Označena je z dogodkom, ki ga sproži. Prehod brez dogodka ali akcije se imenuje avtomatski prehod.

Končno stanje (angl. final state) je izstopna točka diagrama. Sklenjeni diagrami (angl. closed loop) nimajo končnega stanja, saj je življenjska doba objekta v diagramu vezana na življenjsko dobo celotnega sistema, katerega podsistem je diagram stanj.

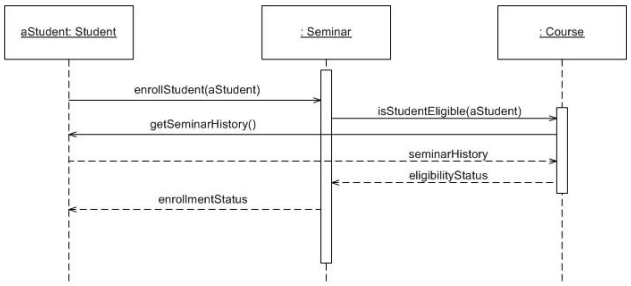


Slika 2.6: Diagram stanj [31].

Diagram zaporedij

Ta diagram opisuje, v kakšnem zaporedju sporočila potujejo med objekti in akterji v diagramu, ki so aktivni le, ko je to potrebno oziroma ko z njimi želi komunicirati drug objekt. Dogodki so predstavljeni v kronološkem vrstnem redu.

Pri razvoju programske opreme so takšni diagrami uporabljeni za predstavitev različnih objektov (tudi subjektov) in komunikacije med njimi. Pri tem pa ne gre za predstavitve vrste komunikacije, temveč le za kronološki vrstni red izvajanja dogodkov (slika 2.7).



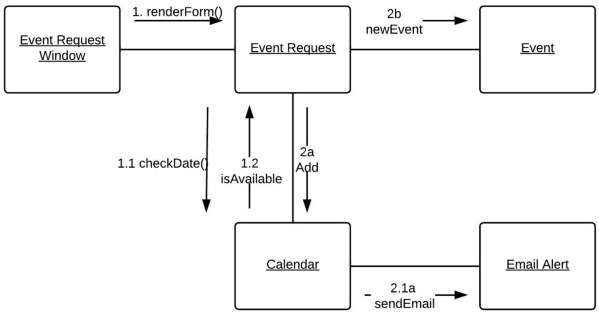
Slika 2.7: Diagram zaporedij [33].

Komunikacijski diagram

Glavni gradnik tega diagrama so sporočila, ki se prenašajo med objekti in akterji. Ideja je podobna kot pri diagramu zaporedij, vendar pa kronološki vrstni red ni pomemben. To pomeni, da diagrama ni potrebno graditi od vrha proti dnu, temveč lahko gradnike in akterje postavimo kamor koli v prostor in med njimi le pošiljamo oštevilčena sporočila. S sledenjem oštevilčenih sporočil dobimo vrstni red izvajanja dogodkov (slika 2.8).

2.1.3 Diagrami obnašanja

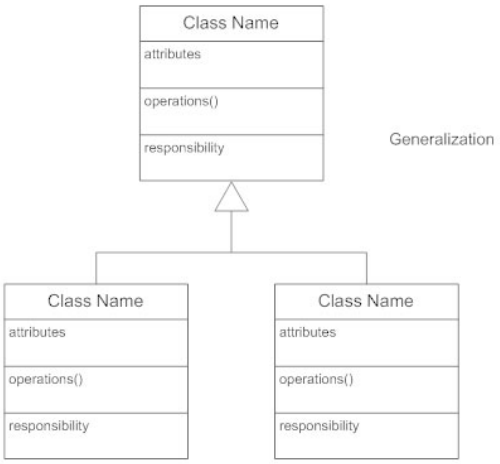
Diagrami predstavljajo obnašanje dinamičnih objektov v sistemu skozi čas ter kako obnašanje spreminja sistem [22, 42, 24].



Slika 2.8: Komunikacijski diagram [35].

Razredni diagram

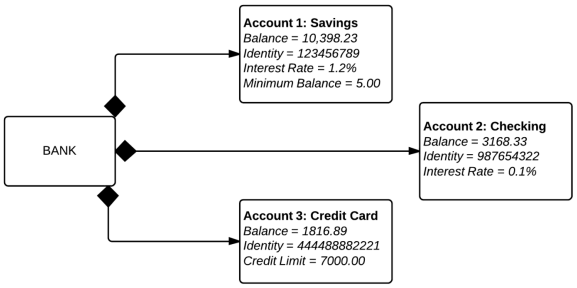
Ta diagram se najpogosteje uporablja pri razvoju programske opreme in je tudi najbolj razširjen. Večina današnjega razvoja namreč temelji na objektno usmerjenem programiranju. Zato je uporaba razrednega diagrama primerna rešitev, saj le-ta temelji na razredih (angl. class) in pa relacijah med njimi. Razredni diagram tako vsebuje ime razreda, atribute razreda in operacije oziroma metode (slika 2.9).



Slika 2.9: Razredni diagram [36].

Diagram objektov

Razredni diagram predstavlja abstraktno strukturo, diagram objektov pa v nasprotju s tem predstavlja instanco te strukture. Gre za statičen objekt, ki se v sistemu pojavi v nekem stanju in s konkretnimi podatki. Razvijalci uporabljajo diagram objektov kot drugo nivojsko preverjanje, saj lahko razredni diagram z uporabo diagrama objektov uporabimo v praktičnem primeru in tako dobimo boljšo predstavitev oziroma potrditev, da smo naše razrede zasnovali pravilno (slika 2.10).



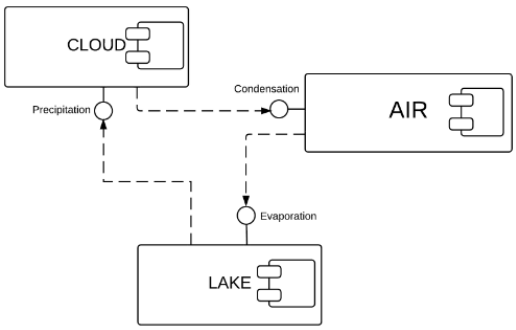
Slika 2.10: Diagram objektov [27].

Diagram komponent

Ta diagram med seboj povezuje manjše v večje komponente v celoten sistem. Prikazuje, kako so med seboj povezane in kako med seboj komunicirajo, ter prek kakšnih kanalov. Te komponente so lahko objekti v trajanju izvajanja (angl. runtime), izvršne datoteke (angl. executable) ali pa kar v obliki izvorne kode (slika 2.11).

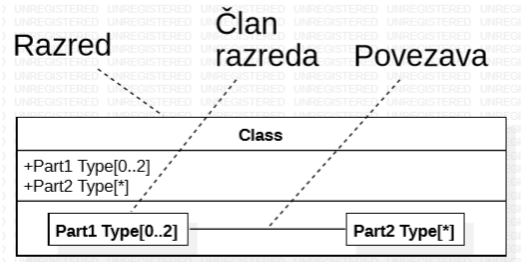
Sestavljen strukturni diagram

To obliko diagrama je vpeljala verzija UML 2.0 in je relativno nova. V praksi je redko uporabljena, saj je namenjena v zelo specifične namene. Sestavljen strukturni diagram je vrsta razrednega diagrama in diagrama komponent, ki



Slika 2.11: Diagram komponent [26].

se uporablja pri zelo natančnem opisu razreda. Prikazuje, kako so med seboj povezani deli znotraj razreda (slika 2.12).



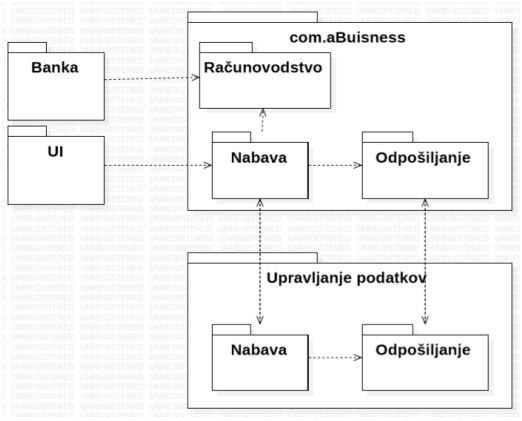
Slika 2.12: Sestavljen strukturen diagram [37].

Diagram sklopov

Diagram sklopov je vrsta vsebovalnika (angl. container), v katerem se lahko nahajajo zgoraj našteti diagrami. Diagram sklopov prikazuje povezave in njihove odvisnosti. Lahko ga tudi gnezdimo in tako zgradimo celotno, zahtevno infrastrukturo, kjer pa podrobne specifikacije niso pomembne (slika 2.13).

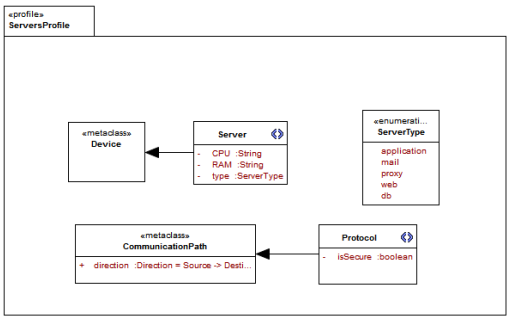
Diagram profilov

Ta diagram omogoča, da projektu dodamo posebne lastnosti (kot so npr. platforma, na kateri bo programska oprema delovala) ter druge stereotipe,



Slika 2.13: Diagram sklopov [29].

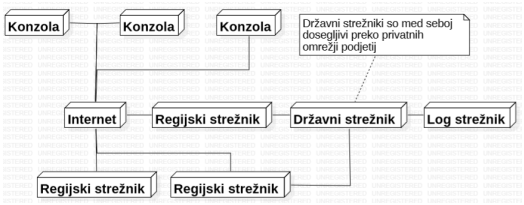
ki se nanašajo na določeno okolje oziroma domeno. Tako lahko še dodatno izboljšamo naš UML diagram in kritičnim odsekom specifične domene posvetimo dodatno pozornost pri razvoju (slika 2.14).



Slika 2.14: Diagram profilov [28].

Umestitveni diagram

Ta diagram se uporablja za simuliranje razvite programske opreme v delovanju s strojno opremo. Diagram se sestoji iz izdelkov (angl. artifacts), ki predstavljajo programsko opremo, in vozlišč (angl. nodes), ki predstavljajo strojno opremo (slika 2.15).



Slika 2.15: Umestitveni diagram [38].

2.1.4 Zaključno mnenje o UML

Izmed celotnega nabora diagramov UML se v praksi najpogosteje uporablja le nekaj izbranih. Ostali so preveč specifični in tako niso uporabni v splošne namene. Kot možen izbor za modeliranje poslovnega procesa sta za nas najprimernejša diagram aktivnosti in diagram stanj.

2.2 Notacija BPM

Notacija za modeliranje poslovnih procesov (angl. Buisness Process Modeling Notation) ali BPMN [40, 9] je vrsta diagrama poteka, ki vsebuje podroben popis vseh objektov in subjektov, ki nastopajo v procesu. Kot taka prikazuje akcije in upravljanje s podatki. Namen modeliranja je optimizacija ter lažje kasnejše prilagoditve in dopolnitve obstoječega modela. Obstaja šest razlogov za modeliranje poslovnega procesa [15]:

1. razumevanje ključnih delov procesa,
2. usmeritev k izdelavi primernega informacijskega sistema, ki bo pripomogel pri poslovanju,
3. optimizacija trenutnega poslovnega procesa,
4. prikaz novosti v procesu,
5. preizkušanje novih poslovnih modelov ter

6. prepoznavanje poslovnih delov, ki niso del jedra procesa in jih lahko upravljamo od zunaj.

2.2.1 Zgodovina notacije BPM

BPMN je bil razvit s strani organizacije BPMI (angl. Buisness Process Modeling Initiative). Leta 2005 je bil predstavljen organizaciji OMG z namenom standardizacije. OMG je prevzela nadaljnji razvoj in leta 2011 izdala BPMN verzije 2.0, ki je vsebovala dopolnjen nabor simbolov (predvsem za izvajanje vzporednih aktivnosti), predstavitev neprekinljivih dogodkov, odločitvenih poti, delitev procesov na podprocese in druge dodatke.

Namen notacije je preprosta predstavitev modeliranega procesa, kar pomeni, da je namenjen vsem subjektom vključenih vanj. Razumljiv naj bi bil tudi poslovnim uporabnikom, ki niso poznavalci. Hkrati pa mora vsebovati dovolj podrobnosti, da ga razvijalec programske opreme zmore sprogramirati.

2.2.2 Gradniki BPMN 2.0

Grafične simbole notacije v grobem delimo v štiri skupine:

1. Objekti v toku (angl. flow objects): dogodki, aktivnosti in skupni prehodi (angl. gateways).
2. Povezovalni objekti (angl. connection objects): zaporedja izvajanj, sporočila ter asociacijske povezave (angl. association).
3. Bazeni in plavalni pasovi (angl. swimlanes).
4. Izdelki (angl. artifacts): podatkovni objekti, skupine, anotacije.

Predstavitev gradnikov

Dogodek sproži akcijo, ki lahko spremeni proces ali pa sproži aktivnost.

Aktivnost je dejanje, ki jo izvede oseba ali sistem.

Skupni prehod je točka odločitve, ki temelji na predpisanih pogojih. Te so lahko zahtevne, enostavne, vzporedne, izključujoče ali vključujoče.

Objekti v toku predstavljajo vrstni red izvajanja aktivnosti.

Sporočila se prenašajo med aktivnostmi.

Asociacijska povezava povezuje izdelke z dogodkom, aktivnostjo ali skupnim prehodom.

Bazeni in plavalni pasovi predstavljajo skupino osebkov v procesu. Bazeni predstavljajo uporabnike v različnih podjetjih, ki pa sodelujejo v istem procesu. Pas je način grupiranja aktivnosti po osebkih, ki jih izvajajo.

Izdelki so informacije, ki jih razvijalci dodajo diagramu, da dosežejo željeno stopnjo natančnosti.

Podatkovni objekti predstavlja podatke, ki so potrebni za izvedbo željene aktivnosti.

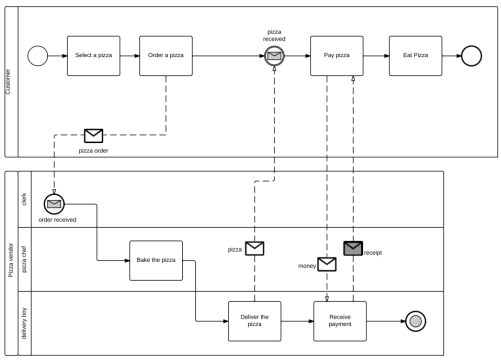
Skupine predstavljajo logično grupiranje aktivnosti, pri tem pa ne spreminjajo diagrama.

Anotacije dodajo dodatno razlago za anotiran del sistema ali podsistema.

Primer diagrama BPM lahko vidimo na sliki 2.16.

2.2.3 BPEL

Jezik za pretvorbo BPM diagramov (angl. Business Process Execution Language) ali BPEL je specifikacija, ki predpisuje, kako preko spletnih storitev (angl. web services) uporabljati sam diagram. BPEL je bil standardiziran leta 2004 in je ena izmed možnosti uporabe izrisanega modela [39].



Slika 2.16: Diagram BPM [8].

2.2.4 Zaključno mnenje o BPMN

BPM notacija je bila razvita z drugačnim namenom kot UML, zato je bolj specifična. V celoti je podrejena potrebam načrtovalcev poslovnih procesov, zato je v druge namene praktično ni smiselno uporabljati. Po predstavitvi obeh notacij lahko vidimo, da sta si diagram aktivnosti iz notacije UML in BPMN zelo podobna, zato ju bomo v naslednjem poglavju primerjali ter predstavili ključne razlike med obema.

2.3 Boost Meta State Machine

Pri končni izvedbi si bomo pri delih diagrama, ki jih ne bo mogoče avtomatsko pretvoriti v programsko kodo, pomagali z modulom MSM (Meta State Machine) [2] iz knjižnice Boost. Pristop temelji na meta programiranju, kar se v C++ programskem jeziku izvede s pomočjo predlog (angl. template). MSM modul je zapisan v zaglavnih datotekah (angl. header files), zato ga ni potrebno posebej prevajati in povezovati. Dovolj je le, da v program vključimo potrebne zaglavne datoteke. Vseeno pa modul zahteva, da se program prevede s prevajalnikom, ki podpira vsaj standard C++14. Za prevajanje bomo tako uporabili prevajalnik MinGw [12].

2.3.1 Gradniki Boost MSM

Diagram stanj (angl. state machine) je model, ki opisuje obnašanje sistema. Sestavljen je iz končnega števila stanj in prehodov.

Stanje je objekt, ki predstavlja trenutno stanje sistema. Lahko drži podatke, ob prehodu v ali iz njega izvede neko akcijo, ali pa izvede notranjo povezavo, ki ne sproži prehoda.

Prehod se nahaja med dvema aktivnima stanjema in ga sproži dogodek. Nanj lahko vežemo akcijo in/ali varovanje (angl. guard). Prehod se zgodi le, ko to dovoli varovanje, takrat pa se sproži tudi akcija.

Notranji prehod (angl. internal transition), se sproži v aktivnem stanju in ne preide v naslednje stanje.

Objekti se nahajajo v dveh imenskih prostorih (angl. namespace). To sta `msm::front` in `msm::back`, saj je tudi diagram stanj sestavljen iz dveh delov. Prvi je sprednji (angl. frontend) del, kjer definiramo objekt z vsemi stanji, prehodi, akcijami in varovanji. Podamo ga zalednemu delu (angl. backend), da ga ta nato uporablja.

Poglavje 3

Primerjava UML diagrama aktivnosti z notacijo BPM

V tem poglavju bomo določili značilnosti, po katerih bomo med seboj primerjali UML diagram aktivnosti in notacijo BPM, ter kriterij ocenjevanja. Vsaka značilnost bo ovrednotena, na koncu pa bomo, glede na rezultate, izbrali primernejšo notacijo. V zaključku poglavja sledi še povzetek prednosti in slabosti obeh.

3.1 Značilnosti primerjave in kriterij

Glavna razlika med diagramom aktivnosti in BPMN je, da prvi uporablja objektno usmerjen pristop pri modeliranju procesa, medtem ko BPMN uporabi procesno usmerjen pristop. To pomeni, da pri diagramu aktivnosti že razmišljamo o končni izvedbi in objekte snujemo v tej smeri. Pri BPMN damo večji poudarek na akcije in podatke v procesu. Pri tem nas ne zanima, kako se bo notacija na koncu izvedla.

Med seboj ju je smiselno primerjati po težavnosti razumevanja simbolov, oziroma koliko predznanja potrebuje poslovni uporabnik brez tehničnega predznanja, da preuči in razume zmodeliran poslovni proces. Primerjali bomo tudi simbole, ki jih uporabimo za predstavitev določene akcije, ko-

liko jih potrebujemo pri prvi in koliko pri drugi notaciji, ter kako zahtevni so. Na koncu bomo oba modela pogledali še z vidika razvijalca, predvsem koliko si lahko pomagamo z njima pri končni izvedbi. Pomembno je tudi, kolikšen delež diagrama lahko pretvorimo v programsko kodo. Ker smo pri BPM notaciji ugotovili, da je zanjo standardiziran jezik za uporabo diagrama preko spletnih storitev, bomo možnost za to vrstno pretvorbo preiskali tudi pri diagramu aktivnosti. Ker pa le-ta vseeno ni primerna rešitev za zastavljene cilje diplomske naloge, bomo večjo težo tudi pri točkovanju posvetili možnosti pretvorbe neodvisno od platforme.

Opisane primerjave tako definiramo v štirih točkah:

1. težavnost razumevanja diagrama,
2. ustreznost elementov v posameznem jeziku,
3. pretvorba v BPEL ter
4. pretvorba v nižje nivojske jezike.

Vsako točko bomo poizkušali ovrednotiti po Likertovi lestvici [11] z ocenami med ena in pet (1-5), kjer 5 pomeni, da je notacija najbolj primerna, 1 pa najmanj.

3.2 Težavnost razumevanja diagrama

Modeliran poslovni proces bo uporabljalo različno število ljudi iz različnih strok. To so recimo procesni analitiki, ki načrtujejo proces, razvijalci programske opreme, ki ga bodo sprogramirali, ter poslovni uporabniki, ki bodo upravljali in opazovali izvajanje procesa. Posebna značilnost BPM notacije je, da poslovni uporabniki, ki so izmed treh omenjenih najmanj tehnično podkovani, ne potrebujejo poglobljenega znanja. Potrebno je le dovolj podrobno poznavanje procesa.

BPMN diagram bi tako morali razumeti vsi vključeni v proces, torej brez poglobljenega predznanja. UML je bil razvit s podobnim namenom, za podporo različnim izvajalcem na projektu ter komunikacijo med njimi.

V raziskavi [15] je bila postavljena hipoteza, da je BPMN diagram bolje razumljiv najmanj tehnično podkovanim uporabnikom v procesu kot UML diagram aktivnosti. Skupini petintridesetih študentov brez predhodnega znanja iz kateregakoli modelirnega jezika so predstavili modeliran proces, katerega poteka prav tako niso poznali. Model so predstavili v ustreznih verzijah obeh standardov. Pri tem je sedemnajst študentov v vpogled dobilo UML diagram aktivnosti, ostalih osemanjst pa diagram BPMN. Vsi študentje so dobili enak vprašalnik z enajstimi drži/ne drži vprašanji, kjer so na podlagi diagrama odgovarjali. Hipoteza je bila ovržena, saj dobljeni rezultati niso dokazovali, da bi bil BPMN lažje razumljiv kot diagram aktivnosti.

Tudi sam sem se med raziskovanjem obeh notacij sproti učil ene in druge. Podobno kot raziskava nisem izkusil, da bi katera bila zahtevnejša za razumevanje kot druga.

3.2.1 Ugotovitev

Na podlagi omenjene raziskave in svojih izkušenj lahko sklepamo, da je težavnost razumevanja BPMN in diagrama aktivnosti približno enaka.

3.3 Ustreznost elementov v posameznem jeziku

Tako pri UML diagramu aktivnosti kot pri BPMN lahko vzorce za predstavitev poteka dela grupiramo v skupine:

- kontrola pretoka,
- pretok podatkov,
- pretok virov in
- obravnavanje izjem.

Kontrola pretoka skrbi za izvajanje opravil in odvisne elemente, ki jih le-ti potrebujejo za izvajanje. Pretok podatkov predstavlja vse podatke v procesu. Pri pretoku virov gre za vse vire procesa ter zato, kako se upravlja z njimi in med njimi razporeja delo. Obravnavanje izjem je pomemben del vsakega dela programske opreme. Tako je tudi pri modeliranju poslovnih procesov pomembno, kako se proces odzove ob nepravilnem delovanju ter kakšni ukrepi morajo biti izvedeni v primeru določene napake.

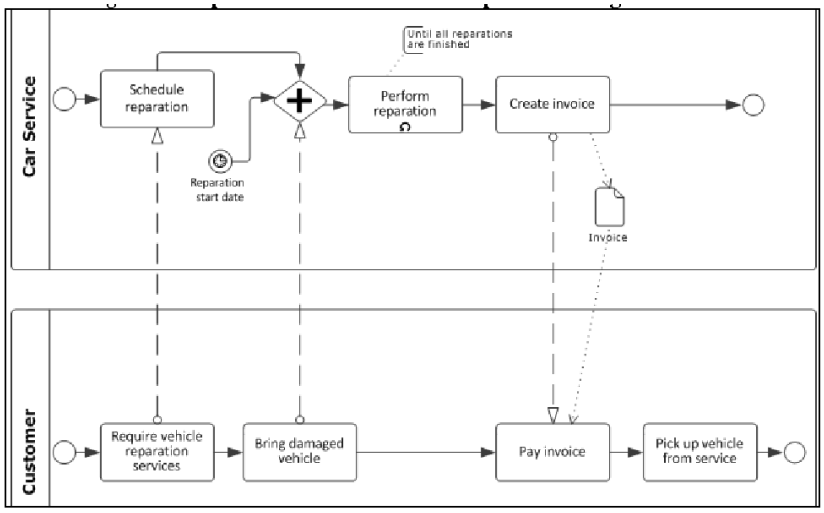
Tako UML diagram aktivnosti kot BPMN za kontrolo pretoka in pretok podatkov zagotavljata podobne rešitve, vendar pa sta oba zelo omejena pri zagotavljanju rešitve za predstavitev virov v diagramu in obravnavo izjem.

Za primerjavo sta bila izdelana BPMN (slika 3.1) in diagram aktivnosti (slika 3.2) za primer preprostega procesa popravila avtomobila pri mehaniku. Proces se začne z rezervacijo termina. Na dan popravila stranka pripelje vozilo k mehaniku, ta pa izvede popravilo in izda račun. Pred prevzemom vozila mora stranka plačati račun.

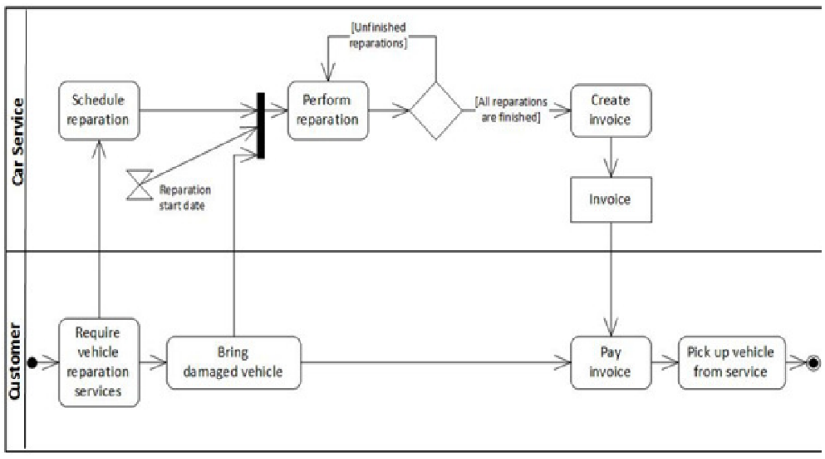
3.3.1 Ugotovitev

Na podlagi primerjav elementov UML diagrama (slika 3.1) in BPMN diagrama (slika 3.2) lahko zaključimo, da so si ti med seboj podobni, vendar pa je UML diagram aktivnosti na nekaterih mestih zahtevnejši. Posamezne dele poslovnega procesa lahko v BPMN modeliramo z uporabo enega simbola, medtem ko moramo pri diagramu aktivnosti uporabiti več simbolov iz nabora. Razlog za to je, da BPMN ne uporablja posebnih simbolov za predstavitev vsake posamezne komponente poslovnega procesa. Z enim simbolom tako lahko predstavimo zelo zahtevne informacije. Diagram aktivnosti tega ne podpira, zato moramo zahtevnejše elemente procesa zmodelirati z osnovnimi simboli.

To je razvidno iz predstavitve aktivnosti *Opravi popravilo (angl. Perform reparation)*. V BPMN diagramu (slika 3.1) se uporabi le en simbol z anotacijo, da gre za zanko. Pri diagramu aktivnosti (slika 3.2) pa je ista aktivnost predstavljena z uporabo simbola aktivnosti, odločitvenega vozlišča,



Slika 3.1: BPMN diagram za proces popravila avtomobila [10].



Slika 3.2: UML diagram za proces popravila avtomobila [10].

ter dveh prehodov, kjer eden predstavlja povratno zanko, drugi pa opravljeno popravilo.

3.4 Pretvorba v BPEL

Jezik za izvajanje poslovnih procesov BPEL zna ob ustrezni preslikavi simbolov poganjati BPM modele. Za tako preslikavo obstajajo orodja, kljub temu pa celotnega modela ne moremo popolnoma avtomatizirano pretvoriti v BPEL. V BPMN lahko opravila povežemo med seboj v kakršnem koli vrstnem redu ali obliki, medtem ko BPEL podpira samo enosmerne povezave, prav tako pa ne razume zank. Specifikacija preslikave BPMN v spletne storitve BPE (angl. Mapping BMPN Models to WS-BPE) tako loči dve vrsti preslikave (osnovno in napredno), kjer pri napredni popolnoma avtomatizirana preslikava ni mogoča.

Čeprav UML standard (OMG, 2011b) ne opisuje pretvorbe UML diagrama aktivnosti v BPEL, pa danes kljub temu obstajajo raziskave in orodja, ki to omogočajo [1, 14].

3.4.1 Ugotovitev

Popolnoma avtomatizirana pretvorba BPM diagrama v BPEL ni mogoča v primerih, ko so diagrami zahtevnejši. Čeprav ni del standarda, pa je mogoča tudi pretvorba UML diagrama aktivnosti v BPEL [1, 14].

3.5 Pretvorba v nižjenivojske jezike

Za pretvorbo BPMN v programsko kodo nižjenivojskega programskega jezika (recimo C++) obstajajo rešitve v obliki tokovnih pogonov (angl. workflow engine) in druge odprtokodne izvedbe. Taka rešitev pa ni primerna za zastavljen problem, saj želimo kodo modela poganjati neodvisno od platforme. Tudi pri UML diagramu aktivnosti je stanje podobno, saj neodvisna pretvorba ni definirana.

3.5.1 Ugotovitev

Pretvorba povsem neodvisno od platforme tako ni definirana v nobenem izmed standardov.

3.6 Povzetek primerjave

UML DA in BPMN smo primerjali na podlagi štirih točk. Po omenjeni raziskavi [10], lahko trdimo, da sta oba diagrama enako zahtevna oziroma nezahtevna za razumevanje pri poslovnih uporabnikih, zato smo oba ocenili s tremi točkami.

Simboli za modeliranje procesa so podobni, vendar pa se zahtevnejše informacije ne da predstaviti z uporabo enega samega simbola v diagramu aktivnosti, medtem ko je to mogoče v BPMN. Iz tega vidika smo diagram aktivnosti ocenili s tremi točkami, medtem ko smo BPMN ocenili z dvema točkama.

BPMN standard opisuje tudi pretvorbo v BPEL jezik, vendar ne popolnoma avtomatizirano, zato smo ga ocenili s tremi točkami. UML v standardu (OMG 2011b) tega ne opisuje, kljub temu pa je taka pretvorba možna, zato smo ga ocenili z dvema točkama [1, 14].

Oba modela sta zelo omejena pri pretvorbi v nižjenivojske programske jezike, zato bi lahko oba ocenili z eno točko. Ne glede na to, je UML bolj pisan na kožo razvijalcem programske opreme in je bolj splošno namenski kot BPMN. Ker si bomo pri končni izvedbi lahko pomagali tudi z drugimi tipi diagramov, ki pa jih je možno pretvoriti v programsko kodo, je za nas iz tega vidika primernejši UML in ga bomo zato ocenili z dvema točkama. V skupnem seštevku rezultatov v tabeli 3.1 je tako za točko boljši UML diagram aktivnosti.

	UML DA	**BPMN**
Težavnost razumevanja diagrama	3	3
Ustreznost elementov v posameznem jeziku	3	2
Pretvorba v BPEL	2	3
Pretvorba v nižje nivojske jezike	2	1
Skupno	**10**	**9**

Tabela 3.1: Točkovna primerjava UML DA in BPMN

3.7 Prednosti

3.7.1 Prednosti BPMN

Vizualizacija celotnega procesa z BPMN lahko odpravi napake pri zasnovi in načrtovanju. Proces lahko načrtujemo iz različnih zornih kotov, pogled imamo na širšo sliko, kjer zraven lahko vključimo še tretje osebe, ki sicer niso del samega procesa, vendar pa lahko njihove akcije znatno vplivajo na potek dogodkov.

Enak model lahko z različno stopnjo podrobnosti uporabljajo vsi vključeni v proces, od vodstva, ki nadzoruje in optimizira proces, do končnih uporabnikov in razvijalcev.

Modeliranje poslovnih procesov v podjetjih se na daljši rok obrestuje, saj model služi tudi kot specifikacija in dokumentacija h končni rešitvi. To olajša delo kasnejšim skrbnikom.

BPMN model lahko pretvorimo v BPEL. To je jezik, ki temelji na spletnih storitvah (angl. web services) in poganja naše modele.

3.7.2 Prednosti UML diagrama aktivnosti

Diagram aktivnosti je eden izmed štirinajstih tipov v popularnem standardu UML. Ta se nagiba k objektno usmerjenem modeliranju in se zato lahko določene stvari reši že pri načrtovanju in ne pri sami izvedbi. Prav tako je namenjen različnim akterjem v procesu, poglobljeno znanje za razumeva-

nje ni potrebno. Služi kot dokumentacija in specifikacija, iz njega pa lahko generiramo različne izhode. Najpogosteje je to izvorna koda.

3.8 Slabosti

3.8.1 Slabosti BPMN

Diagram lahko postane obsežen in zahteven. Če je proces modeliran napačno, trpijo vsi v proces vključeni uporabniki, hkrati pa podjetju prinaša izgubo, saj ga je potrebno popraviti. Prav tako za modeliranje poslovnega procesa potrebujemo strokovnjaka.

3.8.2 Slabosti UML diagrama aktivnosti

UML DA lahko postane obsežen in bolj zahteven še hitreje kot BPMN, saj za modeliranje zahtevnejših informacij potrebujemo več simbolov kot pri BPMN notaciji.

Samo UML DA v večini primerov ne zadostuje. Navadno je proces potrebno zmodelirati z uporabo več diagramov, saj sam diagram aktivnosti ne vsebuje dovolj podrobnosti, ki jih za svoje delo potrebuje razvijalec programske opreme.

3.9 Ugotovitev

Po predstavljenih ugotovitvah smo se za modeliranje zastavljenega problema odločili uporabiti UML diagrama aktivnosti, saj nam kot razvijalcem programske opreme objektno modeliranje poslovnega procesa pripravi podlago tudi pri programiranju.

Problem bomo tako modelirali z uporabo UML diagramov in na delih, kjer nam bo to omogočeno, generirali programsko kodo C++, katero bomo vključil v končni program. Za dele, kjer kode ne bo mogoče generirati iz modelov, bomo uporabili programsko knjižnico Boost in modul MSM.

Poglavje 4

Naloga

V poglavju bomo definirali poslovni proces likvidacije vhodnega računa, ki ga bomo uporabili za konkreten prikaz delovanja. V poslovni proces sodi sam račun s predpisanimi podatki, možne operacije, ki jih izvajamo nad njim, ter stanja, v katerih se nahaja skozi celoten proces. Zraven sodijo tudi vsi uporabniki, ki z njim upravljajo in izvajajo operacije glede na pooblastila in stanje v katerem se le-ta nahaja. V nadaljevanju bo opisana izvedba, kjer bo z uporabo konkretnega primera definiranega procesa podrobneje predstavljen tudi modul MSM in njegovo delovanje. Na koncu poglavja je prikazano še delovanje aplikacije, kjer simuliramo vhodne akcije uporabnikov in tako vodimo potek procesa likvidacije vhodnega računa od začetka do konca.

4.1 Proces likvidacije vhodnega računa

Oseba v podjetju A naroči dejavnost podjetja B, za katero podjetje B izstavi račun. Ker podjetje A posluje elektronsko, je tudi izstavljen račun v takšni obliki. Račun oseba iz podjetja A prejme v svojo poslovno aplikacijo, kjer imajo do njega dostop tudi drugi uporabniki. Ta vsebuje različne podatke, nad njim pa morajo biti izvedene operacije s strani različnih uporabnikov, preden se račun knjiži in plača. Podatki na računu so:

- enolična številka računa,

- datum opravljene storitve,
- datum izdanega računa,
- datum zapadlosti računa,
- skupni znesek,
- razdelilnik (tabela s stolpcema stroškovno mesto in znesek) in
- saldo (razlika med vsoto zneskov v razdelilniku in skupnim zneskom).

Operacije, ki jih uporabniki s pravicami lahko izvedejo na računu v točno določenem stanju, so:

- potrdi račun,
- podpiši račun ali
- knjiži račun.

Delovna mesta uporabnikov, ki bodo rokovali z računom, so:

- vodja,
- nabava,
- finance ter
- vodstvo.

Ko račun prispe v podjetje, ga prevzame *Uporabnik 1*, ki je zaposlen v oddelku nabave in je tudi naročnik storitve podjetja B. *Uporabnik 1* ve, kaj je naročil, zato preveri račun.

Vsi podatki (enolična številka računa, datum opravljene storitve, datum izdanega računa, datum zapadlosti računa, skupni znesek, razdelilnik, saldo) morajo biti izpolnjeni. V nasprotnem primeru, ta takoj zavrne račun.

Če pa so podatki izpolnjeni, mora *Uporabnik 1* preveriti še, ali so veljavni. Tukaj velja, da enolična številka računa še ne sme obstajati med že prejetimi

in knjiženimi računi podjetja. Poleg tega je datum izdanega računa manjši ali enak datumu dneva, ko je bil račun poslan v podjetje. Prav tako mora biti datum opravljene storitve manjši ali enak datumu izdanega računa, datum zapadlosti računa pa mora biti večji ali enak datumu računa, če le-temu prištejemo osem dni.

Če podatki niso veljavni, se račun zavrne, drugače pa se ta prestavi v stanje **prejeto**, kjer nanj čaka *Uporabnik 2*, ki predstavlja zaposlenega v oddelku finance. Ta nato preveri, če je saldo na računu enak nič. Če je temu tako, ga *Uporabnik 2* lahko potrdi, drugače pa se račun zavrne.

Če se za potrditev odloči, se preveri, ali je skupni znesek na računu manjši ali enak pet tisoč evrov. Če je temu tako, se račun prestavi v stanje **potrjen račun**. Drugače se račun prestavi v stanje **delno potrjen račun**, kjer nato čaka na odziv *Uporabnika 3* ali *Uporabnika 4*, ki predstavljata oddelek vodstva.

V tem primeru znesek na računu presega neko določeno mejo, zato za potrditev le-tega potrebujemo soglasje osebe z večjimi pravicami in odgovornostjo. To sta *Uporabnik 3* in *Uporabnik 4*. Nekdo izmed niju mora račun pregledati in se odločiti, ali ga bo potrdil ali zavrnil. Če se ga odloči potrditi, gre račun v stanje **potrjen račun**, kjer čaka na akcijo *vodje*. Ta nato podpiše račun, kar pomeni, da ga sedaj lahko *Uporabnik 1* knjiži in bo tudi plačan.

V prikazanem primeru poslovnega procesa likvidacije vhodnega računa imamo hierarhijo uporabnikov, ki upravljajo z objektom `Račun`, ki spreminja stanja v procesu. Uporabniki imajo v procesu različne vloge in pravice. *Uporabnik 1* je naročnik storitve, vendar pa nima pravic, upravljanja s financami v podjetju, zato mora račun posredovati naprej, ljudem na višjih položajih. Račun mu lahko potrdi *Uporabnik 2*, vendar pa v primeru, ko znesek presega določeno mejo, tudi *Uporabnik 2* potrebuje odobritev *Uporabnika 3* ali *Uporabnika 4*, ki sta v hierarhiji še višje.

Vsak račun mora na koncu odobriti oziroma podpisati *vodja* podjetja. Ta ne spremlja vseh naročil, ki jih izdajajo njegovi zaposleni, zato zaupa

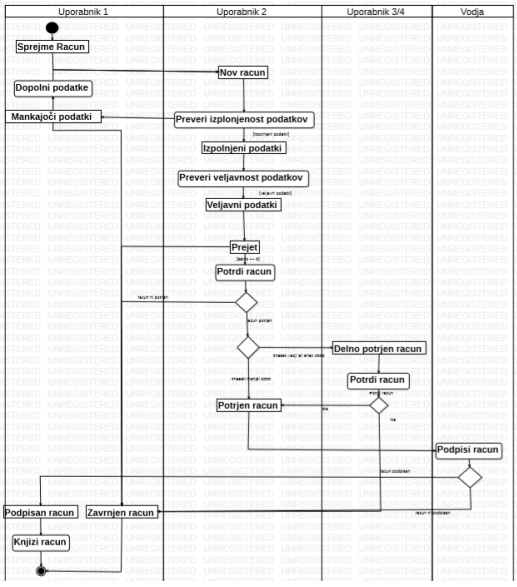
hierarhiji ljudi pod sabo, da je bila storitev na računu res opravljena in so vsi podatki pravilni. To mu omogoča, da *Uporabniku 1* hitro odobri sredstva za izdan račun.

V definiranem poslovnem procesu vedno vemo v katerem stanju se račun nahaja ter kateri uporabniki so z njim že (ali pa še morajo) rokovati.

4.2 Izvedba

Naloge smo se lotil z definiranjem objektov računa in uporabnikov, ki nastopajo v procesu. Nato smo z uporabo diagrama aktivnosti načrtovali model (slika 4.1). Končna različica je bila izrisana z uporabo orodja StarUML [18], ki je na voljo v prosto dostopni poizkusni različici.

Poleg diagrama aktivnosti smo za proces likvidacije vhodnega računa izrisali tudi razredni diagram in diagram stanj, prav tako z uporabo programa StarUML. Z razširitvijo za omenjeni program [19] smo iz razrednega diagrama generirali C++ kodo objektov, ki so kasneje uporabljeni pri izvedbi.



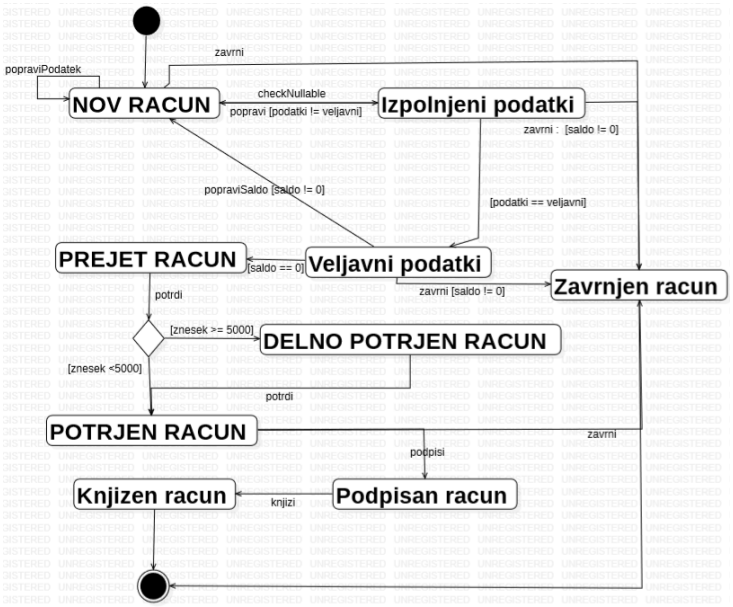
Slika 4.1: Diagram aktivnosti za primer likvidacije vhodnega računa.

4.3 Priprava na izvedbo

V tem delu sta predstavljena konkretna UML diagrama, s katerima smo si pomagali pri končni izvedbi. To je razredni diagram, s katerim smo generirali tudi kodo vseh objektov, ki nastopajo v procesu. Drugi je diagram stanj, katerega smo izdelali za pomoč pri programiranju.

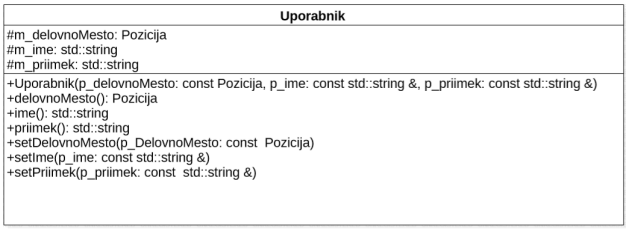
4.3.1 Generiranje razredov

Iz razrednega diagrama smo generirali programsko kodo objektov `Racun`, `Razdelilnik`, `Uporabnik`.



Slika 4.2: Diagram stanj za primer likvidacije vhodnega računa.

Primer definicije razreda v programu StarUML za objekt `Uporabnik` je prikazan na sliki 4.3. Rezultat generiranja kode z razširitvijo [19] za primer objekta `Uporabnik` je viden na sliki 4.4.



Slika 4.3: Razredni diagram za razred `Uporabnik`.

```
#ifndef OBJECTS_UPORABNIK_H
#define OBJECTS_UPORABNIK_H

#include <defs.h>

namespace Objects {

class Uporabnik {
public:
    enum Pozicija
    {
        Proizvodnja = 0,
        Nabava      = 1,
        Finance     = 2,
        Vodstvo     = 3,
        Vodja       = 4
    };

/**
 * @param p_delovnoMesto
 * @param p_ime
 * @param p_priimek
 */
Uporabnik( const Pozicija p_delovnoMesto, const std::string & p_ime, const std::string & p_priimek );

Pozicija delovnoMesto() const;

std::string delovnoMestoToString() const;

std::string ime() const;

std::string priimek() const;

/**
 * @param p_DelovnoMesto
 */
void setDelovnoMesto( const Pozicija p_DelovnoMesto );

/**
 * @param p_ime
 */
void setIme( const std::string & p_ime );

/**
 * @param p_priimek
 */
void setPriimek( const std::string & p_priimek );
protected:
    Pozicija m_delovnoMesto;
    std::string m_ime;
    std::string m_priimek;
};

} // namespace Objects {

#endif //OBJECTS_UPORABNIK_H
```

Slika 4.4: Generirana koda razreda `Uporabnik`.

Vsa koda metod je v `Uporabnik.cpp` datoteki, kjer je razširitev nastavila strukturo, logiko pa je bilo potrebno dodati. Po dopolnitvi vseh generirarnih razredov je bil naslednji korak načrtovanje sprednjega dela MSM diagrama stanj.

4.3.2 Diagram stanj za primer likvidacije vhodnega računa

Ker modul MSM temelji na diagramih stanj, smo diagram aktivnosti (slika 4.1) zmodeliral še z uporabo UML diagrama stanj, katerega končna različica je vidna na sliki 4.2.

4.4 MSM sprednji del

Strukturo sprednjega dela MSM zgradimo iz definicije, ki se nahaja v imenskem prostoru `boost::msm::front::state_machine_def<>`. Če želimo, da jo zaledni del zna uporabljati, mora vsebovati potrebne objekte oz. preobložiti potrebne metode.

Vsebovati mora vsa stanja diagrama, tabelo prehodov med stanji, akcije, ki jih navedemo v tabeli prehodov, ter varovanja.

4.4.1 Definicija strukture `VhodniRacun`.

Struktura z definicijo potrebnih metod in objektov se imenuje `VhodniRacun`. Njena definicija je vidna na sliki 4.5.

Diagram stanj kot vhodni parameter prejme kazalca na objekta `Racun` in `Uporabnik`, ki predstavljata vhodni račun in uporabnika, ki ga je prevzel.

4.4.2 Stanje

Stanje se definira z dedovanjem objekta `boost::msm::front::state<>`. Primer take definicije je viden na sliki 4.6.

```
// frond end state machine definition
struct VhodniRacun : public msm::front::state_machine_def<VhodniRacun>
{

    VhodniRacun( std::shared_ptr<Objects::Racun> p_pRacun, std::shared_ptr<Objects::Uporabnik> p_pUporabnik )
        : m_pRacun( p_pRacun )
        , m_pPrejemnikRacuna( p_pUporabnik )
        , m_pCurrentUser( p_pUporabnik )
    {}

protected:
    std::shared_ptr<Objects::Racun> m_pRacun;
    std::shared_ptr<Objects::Uporabnik> m_pPrejemnikRacuna;
    std::shared_ptr<Objects::Uporabnik> m_pCurrentUser;
```

Slika 4.5: Definicija sprednjega dela MSM diagrama stanj.

```
// State machine states
struct NovRacun : public msm::front::state<>
{
    template<class Event, class FSM>
    void on_entry( Event const &, FSM & )
    {
        spdlog::debug( "VhodniRacun::NovRacun::on_entry: entered state NovRacun" );
    }
    template<class Event, class FSM>
    void on_exit( Event const &, FSM & )
    {
        spdlog::debug( "VhodniRacun::NovRacun::on_exit: exited state NovRacun" );
    }
};
```

Slika 4.6: Definicija stanja `NovRacun` z uporabo modula MSM.

Vsakemu stanju je možno definirati tudi dva dogodka, ki se sprožita ob vstopu ali izstopu iz stanja.

4.4.3 Dogodek

Dogodek je definran kot struktura znotraj strukture `VhodniRacun`. Primer definicije dogodka `popravi` je viden na sliki 4.7. Ta vsebuje podatke o tipu popravka (lahko je eden izmed treh datumov ali pa saldo), ter dejanski popravek, ki je glede na tip lahko niz (angl. string) ali celoštevilski tip (angl. integer).

```
struct popravi
{

    enum Tip
    {
        Storitve   = 0,
        Racuna     = 1,
        Zapadlosti = 2,

        Saldo      = 3
    };

    popravi( const Tip p_tip, const std::string & p_novDatum );
    popravi( const Tip p_tip, const long long int p_saldo );

    ~popravi(){}


    Tip m_tip;
    std::string m_sPopravek;
    long long int m_iPopravek;

};
```

Slika 4.7: Definicija dogodka `popravi`.

4.4.4 Prehodi in varovanja

Prehodi in varovanja so definirani kot metode, ki kot argument prejmejo sklic (angl. reference) na dogodek, kjer metoda varovanja vrne tudi logično vrednost (angl. boolean value). Tako dovoli prehod na povezavi ali pa ga zavrne. Primer je viden na sliki 4.8.

```
//    Guards
    bool allValues( none const & );
    bool allValid( none const & );
    bool allValid( veljavni const & );
    bool saldoIsZero( none const & );
    bool znesekLower( none const & );
    bool znesekHigher( none const & );
    bool checkPopravek( popravi const & p_popravek );
    bool checkUser( potrdi const & p_potrdi );
    bool checkUser( podpisi const & p_podpisi );
    bool checkUser( knjizi const & p_knjizi );

//    transition actions
    void izvediPopravek( popravi const & p_popravek );
    void izvediPotrditev( potrdi const & );
    void izvediPodpis( podpisi const & );
    void izvediKnjizenje( knjizi const & );
```

Slika 4.8: Prehodi in varovanja.

V primeru, da v trenutnem stanju nimamo definiranega prehoda, a se

ta vseeno sproži, lahko definiramo posebno vrsto prehoda, ki se imenuje `notransition` (slika 4.9).

```
template <class FSM, class Event>
void no_transition( Event const & e, FSM &, int state )
{
    std::cout << "V trenutnem stanju ( " << state << " ) dogodek " << typeid( e ).name() << " ni mogoc" << std::endl;
    spdlog::warn( "no transition from state " + std::to_string( state ) + " on event " + typeid( e ).name() );
}
```

Slika 4.9: Nedefiniran prehod.

4.4.5 Tabela prehodov

Tabela prehodov (angl. transition table) mora biti definirana znotraj objekta, ki predstavlja sprednji del diagrama stanj. Definira se jo kot razširitev vektorja `boost::mpl::vector` in se imenuje `transition_table`. Rezultat za primer likvidacije vhodnega računa je viden na sliki 4.10.

```
struct transition_table : mpl::vector<
    //       Start                      Event               Next                Action                    Guard
    //      +--------------------------+-------------------+-------------------+-------------------------+----------------------+
      g_row < NovRacun                 , none              , IzpolnjeniPodatki                           , &v::allValues        >,
      row   < NovRacun                 , popravi           , NovRacun          , &v::izvediPopravek      , &v::checkPopravek    >,
      _row  < NovRacun                 , zavrni            , ZavrnjenRacun                                                    >,
      g_row < IzpolnjeniPodatki        , veljavni          , VeljavniPodatki                             , &v::allValid         >,
      a_row < IzpolnjeniPodatki        , popravi           , NovRacun          , &v::izvediPopravek                           >,
      _row  < IzpolnjeniPodatki        , zavrni            , ZavrnjenRacun                                                    >,
      g_row < VeljavniPodatki          , none              , PrejetRacun                                 , &v::saldoIsZero      >,
      a_row < VeljavniPodatki          , popravi           , IzpolnjeniPodatki , &v::izvediPopravek                           >,
      _row  < VeljavniPodatki          , zavrni            , ZavrnjenRacun                                                    >,
      irow  < PrejetRacun              , potrdi                                , &v::izvediPotrditev     , &v::checkUser        >,
      g_row < PrejetRacun              , none              , Potrjen                                     , &v::znesekLower      >,
      g_row < PrejetRacun              , none              , DelnoPotrjen                                , &v::znesekHigher     >,
      _row  < PrejetRacun              , zavrni            , ZavrnjenRacun                                                    >,
      row   < DelnoPotrjen             , potrdi            , Potrjen           , &v::izvediPotrditev     , &v::checkUser        >,
      _row  < DelnoPotrjen             , zavrni            , ZavrnjenRacun                                                    >,
      row   < Potrjen                  , podpisi           , Podpisan          , &v::izvediPodpis        , &v::checkUser        >,
      _row  < Potrjen                  , zavrni            , ZavrnjenRacun                                                    >,
      row   < Podpisan                 , knjizi            , Knjizen           , &v::izvediKnjizenje     , &v::checkUser        >

    //  +--------------------------+-------------------+-------------------+-------------------------+----------------------+
> {};
```

Slika 4.10: Tabela prehodov.

Zapis v tabelo je predstavljen z objektom `row`, ki sprejme različno število argumentov:

- `row`: sprejme 5 argumentov: začetno stanje, dogodek, naslednje stanje, akcija in varovanje
- `a_row`: sprejme 4 argumente: začetno stanje, dogodek, naslednje stanje in akcija

- `g_row`: sprejme 4 argumente: začetno stanje, dogodek, naslednje stanje in varovanje
- `_row`: sprejme 3 argumente: začetno stanje, dogodek in naslednje stanje

Poleg klasične različice modul zagotavlja tudi posebno vrsto prehoda. To je notranji prehod, ki ima podoben vmesnik kot `row`, vendar pa ne vsebuje argumenta *naslednje stanje*. Notranji prehod se predstavi z objektom `irow`.

4.4.6 Vstopna točka

Vsak veljaven sprednji del diagrama stanj mora definirati tudi vstopno točko. Definira se jo z definicijo tipa `initial_state` (slika 4.11).

```
// the initial state of SM VhodniRacun. Must be defined
typedef NovRacun initial_state;
```

Slika 4.11: Definicija začetnega stanja.

4.5 MSM zaledni del

Struktura mora le še definirati zaledni del, ki izvaja vse definirane akcije, prehode in varovanja. Objekt `VhodniRacun` podamo kot predlogo zalednemu delu (slika 4.12).

```
// Pick a back-end
typedef msm::back::state_machine<VhodniRacun> _vhodniRacun;
```

Slika 4.12: Zaledni del diagrama stanj.

V testnem programu je ustvarjena instanca objekta zalednega dela diagrama stanj, nad katero se kliče metoda `start`. Za procesiranje dogodkov nad instanco se kliče metoda `process_event`. Ta kot vhodni parameter sprejme

sklic na objekt dogodka, ki je definiran v sprednjem delu diagrama. Dogodek glede na tabelo prehodov sproži preverjanje varovanja, prehod med stanji oziroma, kar je definiramo v sami tabeli.

4.6 Primer delovanja

Delovanje diagrama stanj bomo prikazali z uporabo preproste konzolne aplikacije, ki teče v dveh nitih. Prva nit poganja diagram stanj in obdeluje dogodke, medtem ko druga nit sprejema ukaze iz tipkovnice in jih posreduje diagramu v prvi niti. Aplikacija prikazuje potek celotnega poslovnega procesa in z ukazi iz tipkovnice simulira dogodke uporabnikov, ki so definirani v nalogi.

Ko aplikacijo zaženemo se pojavi obvestilo o prispelem računu. Ob inicializaciji vseh potrebnih virov, se kot uporabnik nastavi *Uporabnik 1*. Ta ima možnost prevzeti vhodni račun ali pa ga prepusti nekomu drugemu (slika 4.13).

```
Uporabnik: Uporabnik 1
Delovno mesto: Nabava
------------------------------
Nov Racun, stevilka racuna: 949
------------------------------
[Prevzemi\Prepusti]
```

Slika 4.13: Sprejemanje računa.

Račun prevzame, kar požene diagram stanj v prvi niti in sproži prehod v stanje `NovRacun`. Ob prehodu v stanje se izvede `none` dogodek z varovanjem, ki preveri, ali so vsi obvezni podatki na računu. Če so, se diagram samodejno prestavi v stanje `Izpolnjeni podatki`.

V nasprotnem primeru lahko *Uporabnik 1* izvede dva ukaza. Lahko dopolni manjkajoči podatek in nato ponovno preide v stanje `NovRacun`, kjer se ponovno preveri prisotnost obveznih podatkov, ali pa račun preprosto zavrne.

V tem primeru se proces likvidacije vhodnega računa zaključi in diagram se ustavi.

Iz slike 4.14 je razvidno, da računu manjka *Datum opravljene storitve*, zato diagram ni prešel v naslednje stanje. Za primer simulacije je to mogoče popraviti s klicem ukaza `popravi`.

```
------------------------------
Preverjam prisotnost podatkov
------------------------------
Mankajoci podatki!
Dopolnite podatke ali pa zavrnite racun
=====>[POPRAVI/ZAVRNI]<=====
racun
------------------------------------------------------
| Stevilka racuna: 954
| Datum opravljene storitve:
| Datum izdanega racuna: 31.1.2019
| Datum zapadlosti racuna: 8.2.2019
| Razdelilnik:
|   = Predvidena sredstva: 5000.000000
|   = Nabava materiala: -1000.000000
|   = Storitev: -800.000000
|   = Place izvajalcev: -3000.000000
|   = Prevoz: -180.000000
|   = Mesecni najem: -20.000000
| Skupni znesek: 5000.000000
------------------------------------------------------
```

Slika 4.14: Nepopolni podatki.

```
popravi
..............................
Vrsta popravka
 [1] Datum izdanega racuna
 [2] Datum opravljene storitve
 [3] Datum zapadlosti racuna
 [4] Saldo
 [X] Nazaj
 [Vnesi izbrano operacijo]:
```

Slika 4.15: Dialog vrsta popravka.

V dialogu na sliki 4.15 se izbere vrsto popravka in vnese novo vrednost.

Kot je razvidno iz slike 4.16 se manjkajoči podatek na računu posodobi. Ker popravek v stanju `NovRacun` sproži povratno zanko, se preveri prisotnost vseh podatkov. Tokrat so vsi obvezni podatki prisotni, zato diagram preide v stanje `IzpolnjeniPodatki`.

V stanju `IzpolnjeniPodatki` uporabnik ročno pregleda vse podatke na računu in v primeru veljavnosti izvede dogodek `veljavni`, ki sproži preverjanje varovanja `allValid`. Ta še dodatno preveri veljavnost obveznih podatkov. V primeru, da so podatki veljavni, se diagram prestavi v stanje `VeljavniPodatki` (slika 4.17).

```
.............................
Vrsta popravka
 [1] Datum izdanega racuna
 [2] Datum opravljene storitve
 [3] Datum zapadlosti racuna
 [4] Saldo
 [X] Nazaj
 [Vnesi izbrano operacijo]: 2
.............................
Vnesi popravek za izbran podatek:
1.1.2019
------------------------------
Preverjam prisotnost podatkov
------------------------------
Vsi podatki prisotni - preverite veljavnost!
=====>[VELJAVNI,POPRAVI,ZAVRNI]<=====
racun
-----------------------------------------------
| Stevilka racuna: 949
| Datum opravljene storitve: 1.1.2019
| Datum izdanega racuna: 31.1.2019
| Datum zapadlosti racuna: 8.2.2019
| Razdelilnik:
|   = Predvidena sredstva: 5000.000000
|   = Nabava materiala: -1000.000000
|   = Storitev: -800.000000
|   = Place izvajalcev: -3000.000000
|   = Prevoz: -180.000000
|   = Mesecni najem: -20.000000
| Skupni znesek: 5000.000000
-----------------------------------------------
```

Slika 4.16: Izvedba popravka, prehod v naslednje stanje.

Ob prehodu v stanje `VeljavniPodatki` se sproži `none` dogodek, ki pre-

veri, ali je *saldo* enak nič. Pri izpolnjenem pogoju, se račun prestavi v stanje `PrejetRacun`. Drugače diagram ostane v stanju `VeljavniPodatki`, kjer lahko uporabnik račun zavrne, ali pa izvede popravek nad saldom, ki diagram vrne v stanje `IzpolnjeniPodatki`.

Ko je račun v stanju `PrejetRacun` se kot aktivni uporabnik nastavi *Uporabnik 2*, kar bi v praksi pomenilo, da se račun pojavi v predalčku drugega uporabnika, in čaka na njegovo operacijo. *Uporabnik 2* potem bodisi potrdi račun (slika 4.18), ki se glede na skupni znesek prestavi v stanje `Potrjen` ali pa v stanje `DelnoPotrjen`. V primeru, da je znesek večji ali enak pet tisoč evrov, se račun pojavi v predalčku tudi pri uporabniku *Uporabnik 3* in *Uporabnik 4*, kjer čaka, da nekdo izmed niju izvede potrditev. Ko se to zgodi (slika 4.19), se račun prestavi v stanje `Potrjen`.

```
----------------------------------------
| Stevilka racuna: 949
| Datum opravljene storitve: 1.1.2019
| Datum izdanega racuna: 31.1.2019
| Datum zapadlosti racuna: 8.2.2019
| Razdelilnik:
|   = Predvidena sredstva: 5000.000000
|   = Nabava materiala: -1000.000000
|   = Storitev: -800.000000
|   = Place izvajalcev: -3000.000000
|   = Prevoz: -180.000000
|   = Mesecni najem: -20.000000
| Skupni znesek: 5000.000000
----------------------------------------
veljavni
------------------------------
Preverjam veljavnost podatkov
------------------------------
Veljavni podatki!
=====>[POPRAVI,ZAVRNI]<=====
------------------------------
Preverjam saldo
------------------------------
Veljavni saldo!
=====>[POTRDI,ZAVRNI]<=====
```

Slika 4.17: Potrditev veljavnosti podatkov.

```
------------------------------
Uporabnik: Uporabnik 2
Delovno mesto: Finance
------------------------------
potrdi
[POTRJUJEM RACUN]
------------------------------
Znesek VISJI ali enak dovoljeni meji, racun je delno potrjen
Racun je delno potrjen!
```

Slika 4.18: Delna potrditev računa.

```
------------------------------
Uporabnik: Uporabnik 3
Delovno mesto: Vodstvo
------------------------------
racun
----------------------------------------
| Stevilka racuna: 949
| Datum opravljene storitve: 1.1.2019
| Datum izdanega racuna: 31.1.2019
| Datum zapadlosti racuna: 8.2.2019
| Razdelilnik:
|   = Predvidena sredstva: 5000.000000
|   = Nabava materiala: -1000.000000
|   = Storitev: -800.000000
|   = Place izvajalcev: -3000.000000
|   = Prevoz: -180.000000
|   = Mesecni najem: -20.000000
| Skupni znesek: 5000.000000
----------------------------------------
potrdi
[POTRJUJEM RACUN]
Racun je potrjen!
```

Slika 4.19: Potrditev računa.

Ko je ta potrjen, čaka na podpis. Podpiše ga lahko samo *vodja* podjetja. Ta pregleda račun in se odloči, da bo sredstva odobril, in izvede dogodek `podpisi` (slika 4.21).

Sedaj se *Uporabnik 1*, ki je na začetku prevzel račun, lahko odloči za knjiženje računa, kar stori z izvedbo dogodka `knjizi` (slika 4.20).

```
------------------------------
Uporabnik: Uporabnik 1
Delovno mesto: Nabava
------------------------------
knjizi
[RACUN KNJIZEN]
Racun s stevilko 949 je knjizen!
```

Slika 4.20: Knjiženje računa.

Ko je račun knjižen se proces zaključi. Če je bil le-ta zavrnjen v katerem koli delu procesa, se proces zaključi v trenutku zavrnitve.

```
------------------------------
Uporabnik: Vodja
Delovno mesto: Vodja
------------------------------
racun
----------------------------------------
| Stevilka racuna: 949
| Datum opravljene storitve: 1.1.2019
| Datum izdanega racuna: 31.1.2019
| Datum zapadlosti racuna: 8.2.2019
| Razdelilnik:
|   = Predvidena sredstva: 5000.000000
|   = Nabava materiala: -1000.000000
|   = Storitev: -800.000000
|   = Place izvajalcev: -3000.000000
|   = Prevoz: -180.000000
|   = Mesecni najem: -20.000000
| Skupni znesek: 5000.000000
----------------------------------------
podpisi
[RACUN PODPISAN]
Racun je podpisan!
```

Slika 4.21: Podpis računa.

Poglavje 5

Analiza

V tem poglavju bomo pregledali zastavljene cilje diplomske naloge ter opredelili, do kakšne mere so bili ti izpolnjeni. Podrobna predstavitev in primerjava obeh standardov za načrtovanje poslovnih procesov je že bila predstavljena v poglavjih 2 in 3, zato bo tukaj izpuščena. Osredotočili se bomo na izvedbo zastavljene naloge in razloge za uporabo diagramov stanj. Nato sledi še analiza modula MSM s predstavitvijo prednosti in slabosti. Na koncu se bomo posvetili še razlogom za izbiro modelno usmerjenega razvoja ter njegovim omejitvam.

5.1 Zastavljeni cilji

V podpoglavju 4.2 je izveden poslovni proces likvidacije vhodnega računa. Vanj so vključeni uporabniki in račun, s katerim upravljajo. Celoten proces je vizualiziran z UML diagramom aktivnosti (slika 4.1), kateri je predstavljen še z UML diagramom stanj (slika 4.2). Tega smo z uporabo modula MSM iz knjižnice Boost pretvorili v programsko kodo C++ in jo vključili v testni program, s katerim smo testirali pravilnost delovanja (poglavje 4.6).

Namen testne aplikacije je bil predvsem prikaz pravilnosti delovanja pretvorjenega modela z uporabo nižjenivojske knjižnice Boost, s katero smo zagotovili, da se aktivnosti v modelu na sliki 4.1 izvajajo v pravilnem vr-

stnem redu. Preverili smo tudi, da so prehodi, ki v trenutnem stanju niso definirani, načrtno spregledani, kar pomeni, da so operacije nad objektom račun izvedene s strani pooblaščenih uporabnikov.

5.2 Razlogi za razvoj z uporabo diagramov stanj

Poslovni proces je s stališča programerja skupek kode, ki se izvaja glede na različne vrednosti določenih spremenljivk kot so na primer podatki na računu in uporabniki, ki z njim ravnajo. V primeru, da kombinacija podatkov na računu predstavlja *stanje A* in je uporabnik ustrezen (*uporabnik X*) izvedemo del kode. V primeru, da so podatki v stanju B in uporabnik X, izvedemo drugi del kode. V kateremkoli drugem primeru pa izvedemo tretji del kode.

Opisano logiko lahko programer predstavi z uporabo pogojnih stavkov (angl. if statements), kjer kodo pogoji s spremenljivimi vrednostmi.

```
if( stanje == PrejetRacun &&  uporabnik == "uporabnik2"
&&  saldo == 0 ) {
    potrdi( uporabnik );
    if( znesek < 5000 ) {
        stanje = "Potrjen";
    }
    else {
        stanje = "DelnoPotrjen";
    }
}
else if( stanje == Potrjen && uporabnik == "vodja" )
    podpisi();
```

Takšna koda postane neberljiva in zahtevna za vzdrževanje. Diagrami stanj preprečujejo omenjene težave. Za objekt v diagramu vedno vemo, v

katerem stanju se nahaja, kakšne akcije se v tem primeru lahko izvedejo in s strani katerih uporabnikov. S prehodi iz stanja v stanje se lahko spreminjajo tudi informacije na objektu ali pa le-te spreminjajo delovanje sistema, katerega podsistem je diagram.

Koda je z definicijo struktur (kot je prikazano v podpoglavju 4.4) bolj berljiva in lažje prilagodljiva na spremembe, saj se poslovni procesi podjetja lahko tudi spreminjajo. To pa iz stališča programerja ne sme predstavljati prevelikih težav.

S tega stališča je koda razvitega poslovnega procesa likvidacije vhodnega računa pregledna in lahko dopolnjiva. V primeru, da v procesu definiramo novo aktivnost, moramo le-to predstaviti s konkretnimi stanji, prehodi in varovanji, ter v tabelo prehodov dodati nov zapis.

5.3 Modul Boost MSM

Poleg funkcionalnosti, ki smo jih preizkusili z uporabo opisanega modula, bomo v analizo vključili še dejstva, ki jih izpostavljajo njegovi razvijalci in so za končno oceno prav tako smiselna. To je na primer hitrost, ki je brez uporabe katere druge rešitve ne moremo primerjati. Izpostavili bomo tudi lastnosti, ki ločijo modul MSM od podobnih knjižnic, ter preučili možnost uporabe le-tega v konkretni domeni.

5.3.1 Hitrost

Poleg že omenjenega razloga za uporabo diagramov stanj pri razvoju modul MSM ponuja še nekoliko več. Knjižnice za delo z diagrami stanj so počasne in to je lahko eden izmed razlogov, da se programerji odločajo za ročno programiranje. Razvijalci modula MSM zagotavljajo, da bo delovanje letega v večini primerov hitrejše od katere koli ročne izvedbe [6].

Ker Boost ponuja še en modul za delo z diagrami stanj imenovan Boost Statechart [20], so razvijalci izvedli test, kjer so enak diagram stanj izvedli z obema moduloma [7]. Programa sta bila prevedena na dveh računalnikih z

operacijskima sistemoma Windows XP in Ubuntu 8.04. Za prevajanje so uporabili prevajalnik VC9 in GCC verzije 4.2.3. Za testiranje so sprogramirali dva diagrama stanj. Prvi je bil preprost, medtem ko je bil drugi zahtevnejši, kar pomeni, da je vseboval tudi vgnezdene diagrame. Rezultati so naslednji:

VC9	preprost primer se je izvedel 90x hitreje z uporabo modula MSM kot Statechart
	zahtevnejši primer se je z uporabo modula MSM izvedel 25x hitreje
GCC 4.2.3	preprost primer se je z uporabo modula MSM izvedel 36x hitreje
	zahtevnejši primer se je izvedel 19x hitreje z uporabo modula MSM

Iz rezultatov je razvidno, da je modul MSM konkretno hitrejši od modula Statechart. Iz tega sklepamo, da je hitrost modula MSM, upravičeno izpostavljena kot pomembna lastnost pri izbiri orodja za delo z diagrami stanj.

5.3.2 Tabela prehodov

MSM kot prednost predstavlja tudi uporabo tabele prehodov, ki se je z objektno usmerjenim načinom programiranja v nekaterih podobnih knjižnicah izgubila, zato se prehodi nanašajo na stanje. Prehod tako pripnemo na objekt, ki predstavlja stanje, s tem pa jim težje sledimo. V modulu MSM pa so vsi dovoljeni prehodi definirani v tabeli, kjer imamo nad njimi večjo preglednost.

5.3.3 Gnezdenje diagramov

Diagrame lahko poljubno tudi gnezdimo, kar pomeni, da lahko stanje predstavlja vstopno točko v povsem nov diagram stanj. Gnezdeni diagrami se definirajo znotraj sprednjega dela in imajo enako strukturo kot zunanji diagram. Lahko jim dodamo tudi več vstopnih in izstopnih točk ter posebne objekte,

ki si zapomnijo stanje diagrama pred zadnjim izstopom, kar omogoča, da v naslednjem vstopu v gnezden diagram pridobimo podatke iz prejšnje iteracije. Ko je aktivno stanje v gnezdenem diagramu, koda v zunanjem diagramu čaka na dokončanje in izstop izven tega.

5.3.4 eUML

Razvijalci modula razvijajo tudi jezik imenovan eUML [4], ki je trenutno še v eksperimentalnem stanju in na nekaterih sistemih povzroča nepričakovane zaustavitve diagrama. Namen le-tega je lažja in bolj berljiva definicija sprednjega dela diagrama. Jezik podpira razumljivejšo definicijo akcij, prehodov, varovanj, stanj in same tabele prehodov in s tem še zbližuje razvijalce in pa strokovnjake iz domene, katere del je diagram stanj.

eUML podpira dva različna načina vnosa v tabelo prehodov. Prvi je podoben načinu definiranja prehodov z uporabo UML jezika. Drugi je bolj pisan na kožo C++ razvijalcem.

```
T_stanje + dogodek [ varovanje ] / akcija == N_stanje
N_stanje == T_stanje + dogodek [ varovanje ] / akcija
```

`T_stanje` je trenutno stanje medtem ko je `N_stanje` naslednje stanje.

5.3.5 Zaledni in sprednji del

Diagram stanj je v modulu sestavljen iz sprednjega in zalednega dela. V prikazu naloge smo definirali sprednji del in ga podali zalednemu. Vendar pa knjižnica omogoča veliko več kot le to. Z ločitvijo diagrama na dva dela lahko tako prvega definiramo na več načinov. Prvi način je bil uporabljen v nalogi. Druga možnost bi bila definicija sprednjega dela diagrama z uporabo jezika eUML. Oba sprednja dela bi tako lahko podali enakemu zalednemu in ta bi izvajal oba.

Tudi za zaledni del so razvijalci modula priskrbeli aplikativni programski vmesnik [3], kjer ga večina uporabnikov ne potrebuje. Za poznavalce in tiste,

ki želijo modul popolnoma izkoristiti, pa lahko z dostopom do zalednega dela popolnoma prilagodijo in optimizirajo delovanje diagrama stanj za svoje potrebe.

5.3.6 Modelno usmerjen razvoj

Modelno usmerjen razvoj (angl. model driven developement) ali MDD je načrtovanje sistema z modelom (npr. UML) kjer je ta načrtovan na zelo abstraktnem nivoju in osredotočen na domeno in ne na metode izvedbe [13, 6]. Gre za način razvoja, kjer na čim enostavnejši način in s čim manj dela avtomatizirano generiramo kodo. Ta predstavlja program ali pa del programa, katerega lahko poljubno vključimo v večjo celoto.

Pogost problem pri takem načinu razvoja je krožno potovanje (angl. round trip), kjer model pretvorimo v izvorno kodo, le-to pa je mogoče pretvoriti nazaj v model. Razčlenjevalniki (angl. parsers) za tako strukturo programa lahko imajo težave z razumevanjem in je niso sposobni dobro pretvoriti nazaj v model. MSM se razvija tudi v smeri, da bi bila taka razčlenjevanja lažja.

5.3.7 Omejitve

Kljub naštetim prednostim MSM vsebuje tudi pomanjkljivosti. V nalogi niso bile uporabljene vse funkcionalnosti modula, kljub temu pa je prevajanje programa, ki uporablja diagram stanj iz omenjenega modula, počasnejše. Tudi razvijalci modula v dokumentaciji opozarjajo, da lahko pri modelih z več kot osemdeset prehodih prevajanje traja več ur [5]. Pri sami uporabi prevedenega modela razlike v hitrosti ni bilo opaziti.

Poglavje 6

Zaključek

Modelno usmerjen razvoj je verjetno ena izmed metod, ki se bodo razvijale in uporabljale tudi v prihodnosti. Ljudje raziskujejo načine, kako bi to tehniko izboljšali in jo pripeljali do točke, kjer bi lahko model načrtovali z vizualnim orodjem, potem pa zanj zgenerirali programsko kodo, ki je neodvisna od platforme in dovolj optimalna, da se lahko primerja z delom programerjev, ki take modele programirajo ročno.

Težava je predvsem hitrost generiranja kode. Taka rešitev je prepočasna v primeru, da obstaja orodje, ki popolnoma avtomatizirano prevede zahteven model vezan na specifično domeno (kot je primer likvidacije vhodnega računa) in je le-ta popolnoma neodvisna od platforme.

Namen diplomske naloge je bil tudi raziskati pristopa za vizualiziranje procesov, nato pa z uporabo programske knjižnice Boost in modula MSM tak model pretvoriti v programsko kodo.

UML je precej obširnejši in ima drugačen namen kot BPMN. Kljub temu pa smo ugotovili, da imata diagram aktivnosti in notacija za modeliranje poslovnih procesov veliko skupnih točk. Po predstavljenih primerjavah in ugotovitvah smo se odločil za uporabo UML notacije, kjer smo poleg diagrama aktivnosti uporabili še druge notacije, ki so nam pomagale pri končni izvedbi.

Za uporabo modula MSM smo se odločili tudi na podlagi prošnje podjetja,

ki se ukvarja z razvojem platforme za razvoj poslovnih aplikacij, predvsem pa svoje lastne presoje, saj smo po začetnem raziskovanju ugotovili, da je modul namenjen za uporabo na zelo nizkem nivoju. To po eni strani pomeni, da mora razvijalec za veliko stvari poskrbeti sam, po drugi pa ima večjo izbiro in možnosti pri izvedbi. Čeprav smo uporabili le del zmožnosti pristopa, smo prišli do ugotovitev, ki bodo pomagale tudi podjetju.

Za podjetje je pomembna še ena podrobnost. Ker bo platforma namenjena razvijalcem poslovnih aplikacij, bodo ti v večini primerov proces definirali na aktivni platformi. Celoten sprednji del diagrama stanj v modulu MSM mora biti definiran ob prevajanju. To pomeni, da podjetje te rešitve ne bo moglo uporabiti v celoti in bo tako verjetno uporabilo le predstavljene koncepte. Modul bodo prepisali in v platformo vključili na drugačen način.

Z izvedbo zastavljenih ciljev smo zadovoljni, a področje razvoja poslovnih aplikacij ponuja še veliko izzivov. Kljub temu, da se tehnologije in aplikacije selijo na splet in v oblake, pa ponekod del le-teh ostaja na preverjenih, celo zastarelih, a vendar delujočih pristopih.

Nadaljevanje naloge vidimo predvsem v smeri razvijanja generatorjev kode, ki bi kot vhod dobili UML model, nato pa z uporabo modula MSM zgenerirali ustrezni diagram stanj, ki bi ga lahko neodvisno vključili v svojo aplikacijo.

Preizkusili bi tudi jezik eUML, ki je trenutno še nestabilen, vendar pa njegov aplikativni vmesnik omogoča bolj prijazno definicijo diagrama stanj. Prav tako bi z modulom poizusili zmodelirati zahtevnejši proces, kjer bi bilo smiselno uporabiti gnezdene diagrame in več kot 80 prehodov. Na ta način bi v diagram stanj vpeljali zahtevnejše sestavne dele in ga tako poizkusili preobremeniti. Naslednji korak bi tako bil optimizacija zalednega dela diagrama skozi predpisan aplikativni vmesnik, ali pa kar poseg v samo jedro.

V primeru, da bi se naloge lotili še enkrat, bi za boljše rezultate primerjave poiskali še kakšen podoben modul in proces likvidacije vhodnega računa izvedli še z njim. Na ta način bi pridobili boljšo sliko o kvaliteti modula MSM.

Vsekakor bomo pozorni tudi na razvoj modelno usmerjenega pristopa, kjer se išče rešitve, kako tehnologije izpopolniti do take mere, da bo mogoča neodvisna pretvorba modela v programsko kodo in nazaj. Cilj je seveda v tem, da bo pretvorba dovolj optimalna in primerljiva z ročnimi izvedbami.

Literatura

[1] Hayat Bendoukha, Yahya Slimani, and Abdelkader Benyettou. UML Refinement for Mapping UML Activity Diagrams into BPEL Specifications to Compose Service-Oriented Workflows. In *Networked Digital Technologies*, pages 537–548, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[2] Boost Meta State Machine. Dosegljivo: `https://www.boost.org/doc/libs/1_69_0/libs/msm/doc/HTML/index.html`. [Dostopano: 21. 3. 2019].

[3] Boost MSM ackend. Dosegljivo: `https://www.boost.org/doc/libs/1_69_0/libs/msm/doc/HTML/ch03s05.html`. [Dostopano: 21. 3. 2019].

[4] Boost MSM eUML. Dosegljivo: `https://www.boost.org/doc/libs/1_69_0/libs/msm/doc/HTML/ch03s04.html`. [Dostopano: 21. 3. 2019].

[5] Boost MSM limitations. Dosegljivo: `https://www.boost.org/doc/libs/1_69_0/libs/msm/doc/HTML/ch04s04.html`. [Dostopano: 21. 3. 2019].

[6] Boost MSM Preface. Dosegljivo: `https://www.boost.org/doc/libs/1_69_0/libs/msm/doc/HTML/pr01.html`. [Dostopano: 21. 3. 2019].

[7] Boost MSM vs Boost Statechart. Dosegljivo: `https://www.boost.org/doc/libs/1_69_0/libs/msm/doc/HTML/ch04.html`. [Dostopano: 21. 3. 2019].

[8] BPMN diagram (slika). Dosegljivo: https://www.dragon1.com/modeling-languages/bpmn. [Dostopano: 12. 5. 2019].

[9] BPMN version 2.0 specification. Dosegljivo: https://www.omg.org/spec/BPMN/2.0/. [Dostopano: 21. 3. 2019].

[10] Cristina Venera GEAMBASU. BPMN vs. UML Activity Diagram for Business Process Modeling. *Journal of Accounting and Management Information Systems*, 11(4):637–651, December 2012.

[11] Likertova lestvica. Dosegljivo: https://en.wikipedia.org/wiki/Likert_scale. [Dostopano: 21. 3. 2019].

[12] Mingw. Dosegljivo: http://www.mingw.org/. [Dostopano: 21. 3. 2019].

[13] Model driven development. Dosegljivo: https://searchsoftwarequality.techtarget.com/definition/model-driven-development. [Dostopano: 2. 6. 2019].

[14] N. M. F. Mustafa and G. V. Bochmann. Transforming dynamic behavior specifications from activity diagrams to BPEL. In *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*, pages 305–311, Dec 2011.

[15] Daniela Peixoto, Vitor A. Batista, Ana P Atayde, Eduardo P Borges, Rodolfo Resende, Clarindo Isaías, and P S Pádua. A Comparison of BPMN and UML 2.0 Activity Diagrams. 01 2008.

[16] Steve McRobb Simon Bennett and Ray Farmer. *Object-Oriented Systems Analysis And Design Using UML*. McGraw-Hill Education, 2006.

[17] Software design and modeling. Dosegljivo: https://sea.ucar.edu/best-practices/design. [Dostopano: 11. 5. 2019].

[18] StarUML. Dosegljivo: http://staruml.io/. [Dostopano: 21. 3. 2019].

[19] StarUML extension. Dosegljivo: https://github.com/staruml/staruml-cpp. [Dostopano: 21. 3. 2019].

[20] The Boost Statechart Library. Dosegljivo: https://www.boost.org/doc/libs/1_69_0/libs/statechart/doc/index.html. [Dostopano: 21. 3. 2019].

[21] Tipi UML diagramov (slika). Dosegljivo: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/. [Dostopano: 24. 3. 2019].

[22] UML. Dosegljivo: http://www.uml.org/what-is-uml.htm. [Dostopano: 21. 3. 2019].

[23] UML časovni diagram (slika). Dosegljivo: https://www.lucidchart.com/pages/uml-timing-diagram. [Dostopano: 24. 3. 2019].

[24] UML diagram. Dosegljivo: https://tallyfy.com/uml-diagram/. [Dostopano: 21. 3. 2019].

[25] UML diagram aktivnosti (slika). Dosegljivo: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-activity-diagram. [Dostopano: 24. 3. 2019].

[26] UML diagram komponent (slika). Dosegljivo: https://www.lucidchart.com/pages/uml-component-diagram. [Dostopano: 24. 3. 2019].

[27] UML diagram objektov (slika). Dosegljivo: https://www.lucidchart.com/pages/uml-object-diagram. [Dostopano: 24. 3. 2019].

[28] UML diagram profilov (slika). Dosegljivo: https://training-course-material.com/training/UML_Profile_Diagram. [Dostopano: 11. 5. 2019].

[29] UML diagram sklopov (slika). Dosegljivo: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-package-diagram/. [Dostopano: 24. 3. 2019].

[30] UML diagram sodelovanja (slika). Dosegljivo: https://en.wikipedia.org/wiki/Interaction_overview_diagram. [Dostopano: 11. 5. 2019].

[31] UML diagram stanj (slika). Dosegljivo: https://training-course-material.com/training/UML_State_Machine_Diagram. [Dostopano: 24. 3. 2019].

[32] UML diagram uporabe (slika). Dosegljivo: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/. [Dostopano: 24. 3. 2019].

[33] UML diagram zaporedij (slika). Dosegljivo: http://www.agilemodeling.com/artifacts/sequenceDiagram.htm. [Dostopano: 24. 3. 2019].

[34] UML history. Dosegljivo: https://www.slideshare.net/ElizaWrightCarter/history-of-uml. [Dostopano: 21. 3. 2019].

[35] UML komunikacijski diagram (slika). Dosegljivo: https://www.lucidchart.com/pages/uml-communication-diagram. [Dostopano: 24. 3. 2019].

[36] UML razredni diagram (slika). Dosegljivo: https://www.smartdraw.com/class-diagram/. [Dostopano: 24. 3. 2019].

[37] UML sestavljen strukturni diagram (slika). Dosegljivo: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-composite-structure-diagram/. [Dostopano: 24. 3. 2019].

[38] UML umestitveni diagram (slika). Dosegljivo: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-deployment-diagram/. [Dostopano: 24. 3. 2019].

[39] What is BPEL. Dosegljivo: https://searchmicroservices.techtarget.com/definition/BPEL-Business-Process-Execution-Language. [Dostopano: 21. 3. 2019].

[40] What is BPMN. Dosegljivo: https://www.lucidchart.com/pages/bpmn?a=0. [Dostopano: 21. 3. 2019].

[41] What is state machine diagram. Dosegljivo: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-state-machine-diagram/. [Dostopano: 21. 3. 2019].

[42] What is UML. Dosegljivo: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/. [Dostopano: 21. 3. 2019].